A Verified CompCert Front-End for a Memory Model Supporting Pointer Arithmetic and Uninitialised Data

Frédéric Besson, Sandrine Blazy, Pierre Wilke

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A Verified CompCert Front-End for a Memory Model supporting Pointer Arithmetic and Uninitialised Data

Frédéric Besson · Sandrine Blazy · Pierre Wilke

Received: date / Accepted: date

Abstract The CompCert C compiler guarantees that the target program behaves as the source program. Yet, source programs without a defined semantics do not benefit from this guarantee and could therefore be miscompiled. To reduce the possibility of a miscompilation, we propose a novel memory model for CompCert which gives a defined semantics to challenging features such as bitwise pointer arithmetics and access to uninitialised data.

We evaluate our memory model both theoretically and experimentally. In our experiments, we identify pervasive low-level C idioms that require the additional expressiveness provided by our memory model. We also show that our memory model provably subsumes the existing CompCert memory model thus cross-validating both semantics.

Our memory model relies on the core concepts of *symbolic value* and *normalisation*. A symbolic value models a delayed computation and the normalisation turns, when possible, a symbolic value into a genuine value. We show how to tame the expressive power of the normalisation so that the memory model fits the proof framework of CompCert. We also adapt the proofs of correctness of the compiler passes performed by CompCert's front-end, thus demonstrating that our model is well-suited for proving compiler transformations.

Keywords verified compilation · C semantics · pointer arithmetic

Publication history: this article is a revised and extended version of the papers "A precise and abstract memory model for C using symbolic values" and "A concrete memory model for CompCert" published respectively in the APLAS 2014 and ITP 2015 conference proceedings (LNCS 8858 and 9236).

F.Besson
Inria, France
E-mail: Frederic.Besson@inria.fr

S.Blazy
Université Rennes 1 – CNRS – IRISA, France
E-mail: Sandrine.Blazy@irisa.fr

P.Wilke
Yale University, USA
E-mail: pierre.wilke@yale.edu

1 Introduction

Formal verification of programs is usually performed at source level. Yet, a theorem about the source code of a safety critical software is not sufficient. Eventually, what we really value is a guarantee about the run-time behaviour of the compiled program running on a physical machine. The CompCert compiler [24] fills this verification gap: its semantics preservation theorem ensures that when the source program has a defined semantics, program invariants proved at source level still hold for the compiled code. For the C language the rules governing so-called *undefined behaviours* are subtle and the absence of undefined behaviours is in general undecidable. As a corollary, whether the semantic preservation applies or not for a C program is in general undecidable.

To partially alleviate the problem, the formal semantics of CompCert C is executable and it is therefore possible to run the semantics for a given program input and validate a finite prefix of the execution trace against a real execution. Jourdan *et al.* [18] propose a more comprehensive and ambitious approach: they formalise and verify a precise C static analyser for CompCert capable of ruling out undefined behaviours for a wide range of programs. Yet, these approaches are, by essence, limited by the formal semantics of CompCert C: programs exhibiting undefined behaviours cannot benefit from any semantic preservation guarantee. This is unfortunate as there are real programs which sometimes exhibit behaviours that are undefined for the formal semantics of CompCert C and for the C standard. This can be a programming mistake but sometimes this is a design feature. In the past, serious security flaws have been introduced by optimising compilers aggressively exploiting the freedom provided by undefined behaviours [32,9]. The existing workaround is not satisfactory and consists in disabling optimisations known to exploit undefined behaviours.

Another approach consists in increasing the expressiveness of the semantics and ruling out undefined behaviours. For a language like C, ruling out any undefined behaviour is not realistic and would incur a slow down that is not considered to be acceptable. Yet, to be compiled faithfully, certain low-level C idioms that are used in practice require more semantics guarantees than those offered by the existing CompCert C semantics. In the present work, we extend the memory model of CompCert to capture C idioms that exploit the concrete encoding of pointers (e.g. alignment constraints) or access partially uninitialised data structures (e.g. bitfields, see Figure 5). Such properties cannot be reasoned about using the existing CompCert memory model [27,26] because i) the pointer representation is abstract and ii) reading uninitialised data results in an undefined behaviour. One of the key insights of our novel memory model is to delay the evaluation of C operations for which no defined semantics can currently be determined. For this purpose, the semantics constructs symbolic values that are stored in and loaded from memory. One key operation is the *normalisation* primitive which turns, when needed, a symbolic value into a genuine value.

The memory model of CompCert is shared by all the intermediate languages of the compiler: from CompCert C to assembly. It consists of memory operations (e.g. `alloc`, `free`, `store`, `load`) that are equipped with properties to reason about them. The memory model is a cornerstone of the semantics of all the languages involved in the compilation chain and its properties are central to the proof of the semantic preservation theorems. The memory model also defines memory injections, a

generic notion of memory transformation that is performed during compilation passes and expresses different ways of merging distinct memory blocks into a single one. For example, at the C level, each local variable is allocated in a different block of memory. Later on in the compilation chain, the local variables of a given function are represented as offsets in a single block representing the stack frame. This transformation of the memory layout is specified by a memory injection. Reasoning about memory injections is a non-trivial task and their properties crucially depend on the memory model.

In this paper, we extend the memory model of CompCert with symbolic values and tackle the challenge of porting memory transformations and CompCert's proofs to our memory model with symbolic values. The complete Coq development is available online [4]. At certain points in this article, theorems are linked to the online development with a clickable Coq logo . A distinctive feature of our memory model is that pointers are mapped to concrete 32-bit integers, thus we drop the implicit assumption of an infinite memory, that the original CompCert has. This has the consequence that memory allocation can fail. Hence, the compiler has to ensure that the compiled program is using no more memory than the source program. This additional requirement does not impact the code generated by the compiler but complicates the correctness proofs.

This paper describes our work towards a CompCert compiler giving semantics to more programs, hence giving guarantees about the compilation of more programs. In particular, it makes the following contributions:

- We define symbolic values that can be stored in memory, and explain how to normalise them into genuine values that may be used in the semantics.
- We present a formal verification of our memory model within CompCert: we reprove important lemmas about memory operations.
- We prove that the existing memory model of CompCert is an abstraction of our model thus validating the soundness of the existing semantics.
- We adapt the proof of CompCert's front-end passes, from CompCert C to Cminor and, in the process, we extend memory injections which are central to the compiler correctness proof.

The paper is organised as follows. First, Section 2 introduces relevant examples of programs having undefined or unspecified behaviours. Then, Section 3 recalls the main features of CompCert, with a special focus on its memory model. Section 4 defines symbolic values, which are at the very core of our proposed extension, and explains how to *normalise* such symbolic values. Section 5 proposes one particular implementation of the normalisation function, based on an SMT solver, which we use to test our semantics on C programs. Section 6 explains how we use these symbolic values in our new memory model and presents the updated semantics of Clight, an intermediate C-like language simpler than CompCert C. Section 7 reports on the experimentations we have performed, in particular the low-level idioms we have identified when executing programs with our symbolic semantics of C. In Section 8, we reprove the properties of the memory model that are needed to prove the correctness of the compiler's front-end passes. We also show that our new semantics subsumes the existing CompCert semantics. Section 9 presents our re-design of the notion of memory injection that is the cornerstone of compiler passes that modify the memory layout. Section 10 details the modifications in the

correctness proofs of the compiler's front-end passes. Related work is presented in Section 11. Section 12 concludes.

2 Motivation for an Enhanced Memory Model

The C standard leaves many behaviours unspecified, implementation-defined or undefined [16, §3.4]. Unsafe programming languages like C have undefined behaviours by nature and there is no way to give a meaningful semantics to an out-of-bound array access.[1] Yet, certain undefined behaviours of C were introduced on purpose to ease either the portability of the language across platforms or the development of efficient compilers.

Unspecified behaviours are behaviours where the standard leaves a choice between two or more alternatives (e.g. the order in which the arguments of a function are evaluated). The choice may vary from one function call to another. Implementation-defined behaviours are unspecified behaviours where the choice has to be documented by the compiler. For instance, the relative size of numeric types is defined but the precise number of bits is implementation-defined. A cast between pointers and integers is also implementation defined. Undefined behaviours are behaviours for which the standard imposes no requirements. For instance, the access of an array outside its bounds and the access of an uninitialised value[2] are undefined behaviours.

In the presence of undefined behaviours, there can be a gap between what a programmer intended to write and the actual behaviour of the program – especially in the presence of aggressive program optimisations. To illustrate this, consider the simple program in Figure 1 where the naive check `i + 1 > i` is meant to check the absence of overflow. The rationale is that under the assumption that the processor is using two's complement arithmetic, an integer overflow results in a wrap around modulo. Compiled with `gcc` (version 4.9.2) at optimisation levels `-O0` and `-O1`, the program behaves as expected, i.e. it prints `Overflow`. However, at higher optimisation levels, the condition `i + 1 > i` is optimised and transformed into `true`. This optimisation is sound from the compiler's perspective because *a)* if the computation does not overflow, it is obvious that `i + 1 > i`, *b)* if it overflows the C standard stipulates that overflow of signed integers is actually an undefined behaviour and therefore the compiler is allowed to remove the else branch.

```
int main(){
  int i = INT_MAX;
  if (i + 1 > i) printf("No Overflow");
  else           printf("Overflow");
  return 0;
}
```

Fig. 1: A simple program triggering undefined behaviour

[1] Typed languages detect illegal accesses and typically throw an exception.

[2] Except if the value is an unsigned char.

This counter-intuitive optimisation is not correct for CompCert because, unlike the C standard, its developers made the choice to define signed overflow as a wrap-around behaviour. We believe that defining the semantics of real-life C idioms is the way to go to reconcile the programmer's intentions with the actual program's behaviour. In the present work, we go further in that direction and give semantics to low-level idioms such as low-level pointer arithmetic and manipulation of uninitialised data.

2.1 Low-level Pointer Arithmetic

The C standard does not specify the bit-width or the alignment of pointers: those are implementation-defined. In CompCert, pointers are assumed to be 4-byte-wide. We consider, for the sake of the following examples, that `malloc` returns pointers that are 16-byte aligned (i.e. the 4 least significant bits are zeros). Pointer arithmetic, in C as well as in CompCert C, is very limited. Valid operations involving pointers are the addition (or subtraction) of an integer offset to (or from) a pointer, and the subtraction of two pointers pointing to the same object. Certain comparisons are valid: two pointers can be compared for equality (==) and disequality (!=). Other comparisons (<, <=, > or >= operator) are only defined when the pointer arguments point to the same object. In order to perform arbitrary operations over a pointer, it is possible to cast it to an unsigned integer of type `uintptr_t` for which the ISO C standard provides the following specification [16, Section 7.18.1.4].

> *[The type `uintptr_t`] designates an unsigned integer type with the property that any valid pointer to `void` can be converted to this type, then converted back to pointer to `void`, and the result will compare equal to the original pointer.*

We also know from [16, Section 6.3.2.3] that any pointer can be converted to a pointer to `void`.

> *A pointer to `void` may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to `void` and back again; the result shall compare equal to the original pointer.*

Note that this specification is very weak and does not ensure anything if a pointer, cast to `uintptr_t`, is modified before being cast back.

In the rest of the paper, we consider a 32-bit architecture. A pointer fits into 32-bits and we implement `uintptr_t` as a 4-byte unsigned integer. More importantly, we ensure that casts between pointers and `uintptr_t` integers preserve the binary representation of both pointers and integers. In other words, casts between pointers and `uintptr_t` integers are a no-op. In the following, we illustrate how existing low-level C idioms can exploit this specification.

2.1.1 Storing information in spare bits

With this specification of pointer casts, consider the expected behaviour of the code snippet of Figure 2. The pointer `p` is a 16-byte aligned pointer to a heap-allocated integer obtained through `malloc`. Therefore, the 4 trailing bits of its

binary representation are zeros. We can write the binary representation of `p` as `0xPQRSTUV0` where letters `P` to `V` are abstract hexadecimal digits. The last digit of the representation of `p` is `0` because of the 16-byte alignment.

```
char hash(void *ptr);

int main(){
  int *p = (int *) malloc(sizeof(int));
  // p = 0xPQRSTUV0
  *p = 0;
  int *q = (int *) (( uintptr_t ) p | (hash(p) & 0xF));
  // q = 0xPQRSTUVH
  int *r = (int *) ((( uintptr_t ) q >> 4) << 4);
  // r = 0xPQRSTUV0 = p
  return *r;
}
```

Fig. 2: Storing information in spare bits of pointers

Next, pointer `q` is obtained from the pointer `p` by filling its 4 trailing bits with a hash of the pointer `p` (the hash is masked with `0xF` to ensure that it uses only 4 bits). We note `H` the abstract digit corresponding to the hash of `p`. The representation of `q` is exactly that of `p` with the last digit changed to `H`. This pattern is commonly used as a hardening technique (e.g. for secure implementations of `malloc`[3]). Then, pointer `r` is obtained by clearing (using left and right shifts) the 4 least significant bits of `q`, resulting in the binary representation of `r` being equal to that of `p`.

Our model provides semantics to this program, which CompCert does not because of the undefined operations on pointers (`hash`, shifts, bitwise OR/AND).

2.1.2 System call return value

It is common for system calls (e.g. `mmap` or `sbrk`) to return either the value `(void *)`-1 to indicate a failure, e.g. because no memory is available, or a pointer aligned on a page boundary. In two's complement arithmetics -1 is encoded by the bit-pattern `0xFFFFFFFF` and a page aligned pointer is of the form `0xPRSTU000`, assuming that the page size is 4kB. Consider the code of Figure 3 which calls `mmap` to allocate a single character and gets as exit code whether the allocation succeeds. In this particular case, the first argument is `NULL` meaning that `mmap` allocates a fresh memory chunk. The second argument is the size in bytes, here 1, of the allocated region. The other arguments set various properties of the region. They have no impact on the semantics of this particular program and can therefore be ignored. Suppose that the call to `mmap` fails and returns -1. In that case, the condition `(void *)` -1 == `(void *)` -1 holds and the program returns 1. Otherwise, if `mmap` succeeds, the condition `0xPRSTU000` == `0xFFFFFFFF` does not hold and the program returns 0. Again, because we model alignment constraints, we give a meaning to this program.

[3] See "free list utilities" in `http://www.opensource.apple.com/source/Libc/Libc-594.1.4/gen/magazine_malloc.c`

```
int main(){
  char *p = (char*)mmap(NULL, 1,
                        PROT_READ|PROT_WRITE,
                        MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
  return (p == (void*) -1);
}
```

Fig. 3: `mmap` usage

```
unsigned int set(unsigned int p, unsigned int flag) {
  return p | (1 << flag);
}

int isset(unsigned int p, unsigned int flag) {
  return (p & (1 << flag)) != 0;
}

int main() {
  unsigned int status = set(status,0);
  return isset(status,0);
}
```

Fig. 4: Reading the uninitialised variable `status`

2.2 Manipulation of Uninitialised Data

The C standard states that any read access to uninitialised memory triggers undefined behaviours [16, section 6.7.8, §10]:

> *If an object that has automatic storage duration is not initialised explicitly, its value is indeterminate.*

Here, indeterminate means that the value is either unspecified or a *trap representation*. In the C terminology, a trap representation does not correspond to an actual value and reading a trap representation is an undefined behaviour. In case the object may have a trap representation[4], reading the value of a variable before it has been initialised is an undefined behaviour. Our semantics is more permissive and never produces trap representations – this is consistent with the behaviour of standard hardware. In our model, uninitialised memory has a indeterminate arbitrary but stable value. To be more precise, we ensure that reading twice from the same uninitialised memory returns the same result. We show below two idioms that benefit from this more defined semantics.

2.2.1 Flag setting in an integer variable

Consider the code snippet of Figure 4 that is representative of a C pattern found in an implementation of a libC (see Section 7.3). The program declares a `status` variable and sets its least significant bit using the set function. It then tests whether

[4] All types expect `unsigned char` may have trap representations.

```
int main() {
  struct {
    unsigned int a0 : 1;
    unsigned int a1 : 1;
  } bf;
  bf.a1 = 1;
  return bf.a1;
}
```

(a) Bit-fields in C

```
1  struct bfs {
2    unsigned char __bf1;
3  } bf;
4
5  int main(){
6    struct  { unsigned char __bf1;} bf;
7    bf.__bf1 = (bf.__bf1 & ~2U) | ((unsigned int) 1 << 1U & 2U);
8    return (int) ((unsigned int)(bf.__bf1 << 30) >> 31);
9  }
```

(b) Bit-fields in CompCert C

Fig. 5: Emulation of bit-fields in CompCert

the least significant bit is set using the isset function. According to the C standard, this program may have undefined behaviour because the set function reads the value of the `status` variable before it is ever written.

However, we argue that this program should have a well-defined semantics and should always return the value 1. The argument goes has follows: whatever the initial value of the variable status, the least significant bit of status is known to be 1 after the call set(status,0). Moreover, the value of the other bits is irrelevant for the return value of the call isset(status,0), which returns 1 if and only if the least significant bit of the variable `status` is 1. More formally, the program should return the value of the expression (status|(1≪0))&(1≪0) != 0 which simplifies to (status|1)&1 != 0, which evaluates to 1 no matter what the initial binary representation of status might be.

2.2.2 Bit-Fields in CompCert

Another motivation is illustrated by the current handling of bit-fields in CompCert: they are emulated in terms of bit-level operations by an elaboration pass preceding the formally-verified front-end. Figure 5 gives an example of such a transformation. The program defines a bit-field bf with two fields a0 and a1; both 1-bit-wide. The main function sets the field a1 of bf to 1 and then returns this value. The expected semantics is therefore that the program returns 1.

The transformed code (Figure 5b) is not very readable but the gist of it is that field accesses are encoded using bitwise and shift operators. Line 7 can be read as bf.__bf1 = (bf.__bf1 & 0xFFFFFFFD) | 0x2, after simplification and evaluation of compile time constants. The mask with 0xFFFFFFFD clears the second least significant bit of bf.__bf1 and keeps all the other bits unchanged. The bitwise OR with 0x2 sets the second least significant bit. Line 8, the value

of the field is extracted by first moving the field bit towards the most significant bit (bf.__bf1 << 30) and then moving this bit towards the least significant bit (>> 31). The transformation is correct and the target code generated by CompCert correctly returns 1. However, using the existing memory model, the semantics is undefined. Indeed, the program starts by reading the field `__bf1` of the uninitialised structure `bf`. This triggers undefined behaviour according to the C standard. Even though this case could be easily solved by modifying the pre-processing step, C programmers might themselves write such low-level code with reads of undefined memory and expect it to behave correctly. With our model of uninitialised memory, this program has a perfectly defined semantics.

3 CompCert's Memory Model

Our work builds on top of CompCert, a formally verified C compiler. In this section, we give an overview of the design of CompCert. In particular, we provide the necessary background to understand the existing memory model and its properties. Readers familiar with CompCert can freely skip this section. In later sections, we propose a novel memory model which fits the framework of CompCert and therefore provides a similar interface.

3.1 The CompCert Compiler

CompCert [25,27] is a full-fledged industrial-strength C compiler that is programmed and formally verified using the Coq proof-assistant. It transforms CompCert C, a very large subset of C (detailed in Section 3.2) into assembly code for x86, PowerPC and ARM architectures. The compilation is performed through ten intermediate languages, from CompCert C to assembly. Every language is equipped with a small-step formal semantics, formally describing the meaning of each statement and expression construct. The semantics observes behaviours, that we write B. We write $P \Downarrow B$ to indicate that program P exhibits behaviour B. Possible behaviours are normal termination with a finite trace of events τ , divergence (i.e. infinite execution) with an infinite trace of events τ^∞ and going-wrong behaviours with a finite trace of events τ. A program goes wrong if it is in a stuck and non-final state: this is the representation of undefined behaviour. We write `Wrong` for the set of going-wrong behaviours. A behaviour B_2 is said to be an improvement of a behaviour B_1 (written $B_1 \preccurlyeq B_2$) either if B_2 is equal to B_1 or if B_1 is a going-wrong behaviour with trace τ and τ is a prefix of B_2's trace. A program P is *safe* if it does not exhibit going-wrong behaviours:

$$Safe(P) \equiv \forall B, P \Downarrow B \Rightarrow B \notin \texttt{Wrong}$$

The semantics of CompCert C is not deterministic (as C permits different evaluation orders for expressions) and may observe several behaviours for a given program input. CompCert's first transformation on C programs reduces this non-determinism, by choosing an evaluation order. Moreover, CompCert can optimise away run-time errors present in the source program, replacing them by any behaviour of its choice, provided that the behaviour observed before the original

program went wrong is preserved.[5] For these reasons, a compiler pass is said to be *correct* (or semantics preserving) when every behaviour of the compiled program C is an improvement of an allowed behaviour of the source program S. This allows the compiler to replace undefined behaviours with more defined behaviours. This property is called backward simulation and can be stated as follows:

$$\forall B, C \Downarrow B \Rightarrow \exists B', S \Downarrow B' \wedge B' \preccurlyeq B$$

If we restrict ourselves to safe behaviours, the behaviours are exactly preserved: this property is called *safe backward simulation* and can be formally stated as follows:

$$\forall B, B \notin \texttt{Wrong} \wedge C \Downarrow B \Rightarrow S \Downarrow B$$

However, reasoning by induction on the source language is easier than reasoning on the target language, because a single step in the source program corresponds in general to multiple steps in the compiled program. Thus, it is easier to prove a forward simulation instead of a safe backward simulation. A forward simulation states that every safe behaviour of the source program is also a behaviour of the compiled program. Formally, $\forall B, B \notin \texttt{Wrong} \Rightarrow S \Downarrow B \Rightarrow C \Downarrow B$. Provided that the target language is deterministic, a forward simulation argument (easier to prove) is equivalent to a safe backward simulation argument (which we need).

CompCert features various simulations such as the *plus* and *star* simulations which map a single step in the source to multiple steps in the target program. In Figure 6, we depict the simpler so-called *lock-step* simulation which maps a single step in the source to a single step in the target. We show hypotheses as plain lines and conclusions as dashed lines. At the level of program states, a *lock-step* simulation between a source program P_1 written in language L_1 and a compiled program P_2 written in language L_2 can be formally stated as follows: whenever programs P_1 and P_2 are respectively in states S_1 and S_2 that match (according to the relation $\sim$), and P_1 can step from S_1 to a state S_1' with trace τ, then there exists a state S_2' such that P_2 can step from S_2 to S_2' with the same trace τ and S_1' and S_2' match. In the figure, the transition relations are implicitly parameterised by the relevant programs P_1 and P_2.

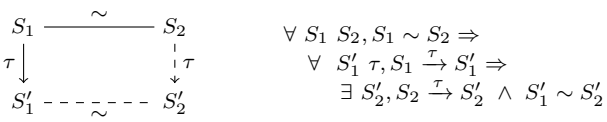


Fig. 6: Lock-step forward simulation diagram

Each of the 15 compiler passes (9 transformations between the 10 intermediate languages plus 6 optimisations) of CompCert is proved correct independently. Then, the simulation theorems are composed together, establishing correctness for the whole compiler.

[5] Note that this preservation is not a requirement of C, but an additional guarantee from CompCert.

3.2 CompCert's Front-End

CompCert is split into a front-end and a back-end. The front-end is architecture-independent, while the back-end is architecture-dependent, i.e. it may use specialized operators only present on some specific hardware. This section introduces the different languages and passes of the front-end of CompCert (see Figure 7).

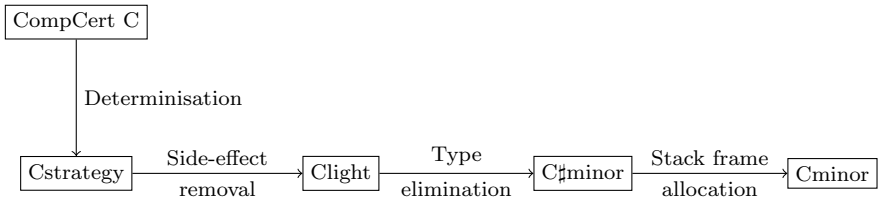


Fig. 7: Architecture of CompCert's front-end

The input language of CompCert's front-end is a large subset of C, called CompCert C, which includes all of MISRA-C 2004 [28] and almost all of ISO C99 [16], with the exceptions of variable-length arrays and unstructured, non-MISRA **switch** statements (e.g. Duff's device). As discussed in the previous section, CompCert C features non-determinism, in particular the order in which the arguments to a function call are evaluated is non-deterministic.

CompCert ships with an interpreter for CompCert C, which allows to test whether a particular execution of a given C program has a defined semantics.

The very first transformation on CompCert C is its determinisation, effectively choosing an evaluation order (or evaluation *strategy*), resulting in the language Cstrategy. Note that the proof of this first pass is necessarily performed as a backward simulation proof, since the forward simulation property does not hold: not every behaviour of the source program has a counterpart in the corresponding determinised compiled program.

Cstrategy programs are then translated into Clight programs. Clight is a subset of C (i.e. any valid Clight program is a valid C program), where side-effects have been pulled out of expressions and made explicit. In Section 6.4, we illustrate our modifications to the semantics using Clight as a representative example.

Clight is then transformed into C♯minor, where all type-information is erased and operations are transformed accordingly. For example, the Clight expression `p+2` where `p` is a pointer to `int` is transformed into the following C♯minor expression: `p+2*sizeof(int)`. The semantics of addition is then simpler in C♯minor because it does not need to reason about the type of its operands, it simply adds an offset to a pointer.

Finally, C♯minor programs are transformed into Cminor programs, where a stack frame is built for every function, and accesses to variables are translated into accesses in the stack frame. This transformation and its proof of correctness are more involved because the memory layout of the program is heavily modified. In Section 10, we adapt the proofs of all these passes for our memory model.

Memory locations:	$loc \ni l$	$::= (b, i)$	(block, integer offset)
Values:	$val \ni v$	$::= \texttt{int}(i) \mid \texttt{long}(l)$	
		$\mid \texttt{float}(f) \mid \texttt{double}(d)$	
		$\mid \texttt{ptr}(l) \mid \texttt{undef}$	
Abstract bytes:	$\texttt{memval} \ni mv$	$::= \texttt{Byte}(b)$	
		$\mid \texttt{Pointer}(b, i, n)$	
		$\mid \texttt{Undef}$	
Memory chunks:	$\texttt{memory\_chunk} \ni \kappa$	$::= \texttt{Mint8signed}$	8-bit integers
		$\mid \texttt{Mint8unsigned}$	
		$\mid \texttt{Mint16signed}$	16-bit integers
		$\mid \texttt{Mint16unsigned}$	
		$\mid \texttt{Mint32}$	32-bit integers or pointers
		$\mid \texttt{Mfloat32}$	32-bit floats
		$\mid \texttt{Mint64}$	64-bit integers
		$\mid \texttt{Mfloat64}$	64-bit floats

Operations over memory states:

$\texttt{alloc}\ m\ lo\ hi = (m', b)$	Allocate a fresh block b with bounds $[lo, hi[$.
$\texttt{free}\ m\ b = \lfloor m' \rfloor$	Free (invalidate) the block b
$\texttt{load}\ \kappa\ m\ b\ i = \lfloor v \rfloor$	Read consecutive bytes (as determined by κ) at block b, offset i of memory state m. If successful, return the contents of these bytes as value v.
$\texttt{store}\ \kappa\ m\ b\ i\ v = \lfloor m' \rfloor$	Store the value v as one or several consecutive bytes (as determined by κ) at offset i of block b. If successful, return an updated memory state m'.
$\texttt{bound}\ m\ b$	Return the bounds $[lo, hi[$ of block b.
$\texttt{size\_chunk}\ \kappa$	Return the size (number of bytes) that κ holds.

Fig. 8: CompCert's memory model

3.3 The Memory Model of CompCert

The memory model of CompCert defines the layout of the memory and the different memory operations. It is shared by all the languages of the CompCert compiler. CompCert uses an abstract block-based model where memory is an infinite collection of separated blocks [26]. Intuitively, a block is an array of bytes that represent values. At the C level, each block corresponds to an allocated variable (e.g. a 32-bit integer is stored in a 4-byte-wide block, an array of 10 characters is stored in a 10-byte-wide block). For languages at a lower level in the compiler chain, this one-to-one correspondence between variables and memory blocks does not hold anymore.

The interface of CompCert's memory model is given in Figure 8. Abstract values (of type *val*) used in the semantics of the CompCert languages (see [27]) are the disjoint union of 32-bit integers (written $\texttt{int}(i)$), 64-bit integers (written $\texttt{long}(l)$), 32-bit floating-point numbers (written $\texttt{float}(f)$), 64-bit floating-point numbers (written $\texttt{double}(d)$), pointers (written $\texttt{ptr}(l)$), and the special value `undef` representing the result of undefined operations or the value of an uninitialised variable. Operations are strict in `undef` i.e. they yield `undef` when one of the operands is `undef`.

$\texttt{ptr}(b,o) \pm \texttt{int}(i)$	$= \texttt{ptr}(b, o \pm i)$	
$\texttt{ptr}(b,o) - \texttt{ptr}(b,o')$	$= \texttt{int}(o - o')$	
$\texttt{ptr}(b,o) \star \texttt{ptr}(b,o')$	$= o \star o'$	when $\star \in \{<, \leq, ==, \geq, >, ! =\}$ and both pointers are weakly valid
$\texttt{ptr}(b,o) == \texttt{ptr}(b',o')$	$= \texttt{false}$	when $b \neq b' \wedge \texttt{valid}(m,b,o) \wedge \texttt{valid}(m,b',o')$
$\texttt{ptr}(b,o) ! = \texttt{ptr}(b',o')$	$= \texttt{true}$	when $b \neq b' \wedge \texttt{valid}(m,b,o) \wedge \texttt{valid}(m,b',o')$
$\texttt{ptr}(b,o) \star \texttt{ptr}(b',o')$	$= \texttt{undef}$	when $b \neq b'$ and $\star \in \{<, \leq, \geq, >\}$
$\texttt{ptr}(b,o) ! = \texttt{int}(0)$	$= \texttt{true}$	when $\texttt{weakly\_valid}(m,b,o)$
$\texttt{ptr}(b,o) == \texttt{int}(0)$	$= \texttt{false}$	when $\texttt{weakly\_valid}(m,b,o)$

Fig. 9: Pointer arithmetic rules in CompCert

A location l is a pair (b,i) where b is a block identifier (i.e. an abstract address) and i is an integer offset within this block. A location (b,i) is valid for a memory m (written $\texttt{valid}(m,b,i)$) if the offset i lies within the bounds of the block b. It is weakly valid (written $\texttt{weakly\_valid}(m,b,i)$) if it is either valid or just one byte past the end of its block. This accounts for a subtlety of the C standard, stating that pointers *one-past-the-end* of an object deserve a particular treatment, namely that they can be compared to the other pointers to this object. This is intended to make looping over an entire array easier, allowing to compare the current pointer to the pointer just *one-past-the-end*.

Example 1 (Valid and weakly valid pointers): Consider a block b with bounds $[0;3[$. Then, pointers $\texttt{ptr}(b,0)$ and $\texttt{ptr}(b,3)$ are valid (and also weakly valid *a fortiori*). Pointer $\texttt{ptr}(b,4)$ is not valid; however it is weakly valid. Pointer $\texttt{ptr}(b,5)$ is neither valid nor weakly valid.

Pointer arithmetic is defined in Figure 9. It reflects the restrictions we described in Section 2.1, that is, the only defined operations are the addition of an integer offset to a pointer, the subtraction of an integer offset from a pointer, and the subtraction of two pointers pointing to the same object, i.e. the same block in CompCert's model. Comparisons are also defined between pointers to the same object. All operations not described are undefined (they return `undef`). Note that, starting from pointer $\texttt{ptr}(b,i)$ it is not possible to reach a pointer to a different block *via* pointer arithmetic, as blocks are separated by construction.

The memory model defines four main memory operations: `load`, `store`, `free` and `alloc`. The `load` and `store` operations are parameterised by a memory chunk κ which concisely describes the number of bytes to be fetched or written, and the signedness of the value. An access at location (b,o) with chunk κ is aligned if $\texttt{size\_chunk}\ \kappa$ divides o[6]. For instance, the size of the chunk `Mint32` is 4 bytes, hence an integer could be accessed with this chunk at offsets that are multiples of 4. These operations are partial, i.e. they may fail e.g. when the access is out of bounds, misaligned, or when the value and the chunk are inconsistent. This is modelled by the option type: we write $\emptyset$ for failure and $\lfloor x \rfloor$ for a successful return of value x. The `free` operation frees a given block. It fails when the given block is not freeable. The `alloc` operation allocates a new block of the requested size. It never fails, thus modelling an infinite memory.

The memory itself is not a direct mapping from locations to values; instead it is a mapping from locations to *abstract bytes* called `memvals`. This allows to reason

[6] It is slightly too strong a condition: a 64-bit float variable only needs to be accessed at addresses that are multiple of 4, not 8.

about byte-level accesses to the memory. A `memval` is a byte-sized quantity that can be one of the following: `Undef` represents uninitialised bytes, `Byte` (b) represents the concrete byte (8-bit integer) b and `Pointer` (b, i, n) represents the n-th byte of the binary representation of the pointer `ptr`(b, i).

3.4 Memory Injections in CompCert

For the front-end, the most difficult proof concerns the compiler pass that modifies the memory layout (i.e. the generation of Cminor from C♯minor). At the C♯minor level, every local variable of a given function is stored in its own block. At the Cminor level, all local variables of a given function are stored in a single stack block, representing its activation record. Memory blocks from the C♯minor program are mapped to offsets in the memory block of the Cminor program. This is shown in Figure 10, where three blocks are merged into a single one.

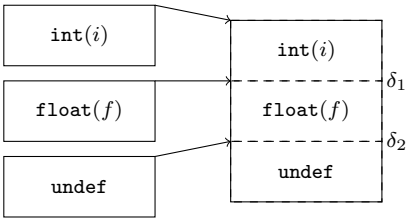


Fig. 10: Injecting local variables into a stack block

The process of merging blocks together is defined in CompCert as a so-called *memory injection*. A memory injection defines a mapping between memories; it is a versatile tool to explain how compiler passes reorganise the memory.

A memory injection is a relation between two memories m_1 and m_2 parameterised by an injection function f : `block` $\rightarrow$ `option location` mapping blocks in m_1 to locations in m_2. The injection relation is defined over values (and called `val_inject`) and then lifted to memories (and called `mem_inject`). The `val_inject` relation is defined inductively in Figure 11.

Rule (1) captures the intuitive semantics of an injection that is depicted in Figure 10. It states that a pointer `ptr`(b_1, i) is in injection with a pointer `ptr`$(b_2, i+\delta)$ if $f(b_1) = \lfloor(b_2, \delta)\rfloor$. Rule (2) states that `undef` is in relation with any value. Finally, rule (3) states that for non-pointer values, the injection is reflexive. The purpose of the injection of values is twofold: it establishes a relation between pointers using the function `f` but it can also specialise `undef` by any defined value.

The relation `memval_inject` lifts `val_inject` to `memval`s and is defined as follows.

1. Concrete bytes are only in injection with themselves.
2. `Pointer` (b, i, n) is in injection with `Pointer` $(b', i + \delta, n)$ when $f(b) = \lfloor(b', \delta)\rfloor$.
3. `Undef` is in injection with any `memval`.

The `mem_inject` relation is built on top of `memval_inject`, but it also includes well-formedness properties. Consider a block b_1 of m_1 injected to a location (b_2, δ) of m_2; the following properties must hold to establish a memory injection between m_1 and m_2.

$$\frac{f(b_1) = \lfloor (b_2, \delta) \rfloor}{\texttt{val\_inject}\ f\ \texttt{ptr}(b_1, i)\ \texttt{ptr}(b_2, i+\delta)}\ (1)$$

$$\frac{}{\texttt{val\_inject}\ f\ \texttt{undef}\ v}\ (2) \qquad \frac{v \neq \texttt{ptr}(b, i)}{\texttt{val\_inject}\ f\ v\ v}\ (3)$$

Fig. 11: `val_inject` in CompCert

- For every valid offset o of b_1, $o + \delta$ must be a valid offset of b_2.
- δ must be properly aligned with respect to the size of b_1.
- For every valid offset o of b_1, the `memval`s at locations (b_1, o) in m_1 and $(b_2, o+\delta)$ in m_2 must be related by `memval_inject`.

The alignment constraint ensures that all aligned accesses remain aligned after the injection, therefore that loads and stores are preserved by the injection. To build a valid memory injection, the function f needs to be injective, i.e. for every pair of disjoint blocks (b_1, b_2), the locations they are injected to do not overlap. The corresponding formal definition is the following:

Definition 1 (`meminj_no_overlap` ):

$$\texttt{meminj\_no\_overlap}\ f\ m : \mathbb{P} := \forall b_1\ b_1'\ \delta_1\ b_2\ b_2'\ \delta_2\ ofs_1\ ofs_2,$$
$$b_1 \neq b_2 \to f(b_1) = \lfloor (b_1', \delta) \rfloor \to f(b_2) = \lfloor (b_2', \delta_2) \rfloor \to$$
$$\texttt{valid}(m, b_1, ofs_1) \to \texttt{valid}(m, b_2, ofs_2) \to (b_1' \neq b_2' \vee ofs_1 + \delta_1 \neq ofs_2 + \delta_2).$$

The memory model provides theorems about the behaviour of memory operations with respect to injections. For example, Theorem 1 states that, starting from two memory states m_1 and m_2 in injection, if a store of a given value v_1 can be performed in m_1 at a location (b_1, o), resulting in a memory state m_1', and if b_1 is injected into location b_2 at offset δ, then a store of a value v_2 (in injection with v_1) can be performed on m_2, resulting in a memory state m_2' such that m_1' and m_2' are in injection.

Theorem 1 (`store_mapped_inject` ):

$$\forall\ f\ m_1\ m_2\ b_1\ b_2\ o\ \delta\ v_1\ v_2,$$
$$\texttt{mem\_inject}\ f\ m_1\ m_2 \to \texttt{store}\ \kappa\ m_1\ b_1\ o\ v_1 = \lfloor m_1' \rfloor \to$$
$$f(b_1) = \lfloor (b_2, \delta) \rfloor \to \texttt{val\_inject}\ f\ v_1\ v_2 \to$$
$$\exists\ m_2', \texttt{store}\ \kappa\ m_2\ b_2\ (o+\delta)\ v_2 = \lfloor m_2' \rfloor \wedge \texttt{mem\_inject}\ f\ m_1'\ m_2'.$$

Similar theorems are proved for the `load`, `alloc` and `free` operations.

4 Symbolic Values and Normalisation

To give a semantics to the C idioms given in Section 2, a direct approach is to have a fully concrete memory model where a pointer is a genuine integer and the memory is an array of bytes. This model is not satisfactory because it prevents abstract reasoning and disables a number of optimisations. Indeed, as Kang *et al.* [19] notice, if addresses are mere integers, any function can *forge* an address and we cannot rely on any isolation property.

Operators:

$$
\begin{array}{rl}
\texttt{op}_1 ::= & \texttt{OpBoolval} \mid \texttt{OpNotbool} \mid \texttt{OpNeg} \mid \texttt{OpNot} \mid \texttt{OpAbs} \\
\mid & \texttt{OpZeroext} \mid \texttt{OpSignext} \mid \texttt{OpRolm} \mid \texttt{OpLoword} \mid \texttt{OpHiword} \\
\mid & \texttt{OpSingleofbits} \mid \texttt{OpDoubleofbits} \\
\mid & \texttt{OpBitsofsingle} \mid \texttt{OpBitsofdouble} \\
\mid & \texttt{OpConvert}(\mathit{tfrom}, \mathit{tto}) \\
\texttt{op}_2 ::= & \texttt{OpAnd} \mid \texttt{OpOr} \mid \texttt{OpXor} \mid \texttt{OpAdd} \mid \texttt{OpSub} \mid \texttt{OpMul} \\
\mid & \texttt{OpDiv} \mid \texttt{OpMod} \mid \texttt{OpShr} \mid \texttt{OpShl} \mid \texttt{OpCmp}(\texttt{cmp}) \\
\mid & \texttt{OpFloatofwords} \mid \texttt{OpLongofwords}
\end{array}
$$

Symbolic values:

$$
\begin{array}{lll}
\mathit{sval} \ni \mathit{sv} ::= & v & \text{value} \\
\mid & \texttt{indet}(l) & \text{indeterminate content of location} \\
\mid & \texttt{op}_1\ \mathit{sv} & \text{unary operation} \\
\mid & \mathit{sv}_1\ \texttt{op}_2\ \mathit{sv}_2 & \text{binary operation}
\end{array}
$$

Evaluation of symbolic values:

$$
\begin{array}{ll}
[\![\texttt{ptr}(b,i)]\!]^{im}_{cm} & = cm(b) + i \\
[\![v]\!]^{im}_{cm} & = v \\
[\![\texttt{indet}(l)]\!]^{im}_{cm} & = im(l) \\
[\![\texttt{op}_1\ \mathit{sv}]\!]^{im}_{cm} & = \texttt{eval\_unop}(\texttt{op}_1, [\![\mathit{sv}]\!]^{im}_{cm}) \\
[\![\mathit{sv}_1\ \texttt{op}_2\ \mathit{sv}_2]\!]^{im}_{cm} & = \texttt{eval\_binop}(\texttt{op}_2, [\![\mathit{sv}_1]\!]^{im}_{cm}, [\![\mathit{sv}_2]\!]^{im}_{cm})
\end{array}
$$

Fig. 12: Semantics of symbolic values

Our approach to improve the semantic coverage of CompCert consists in delaying the evaluation of expressions which, for the time being, do not have an interpretation in terms of values. Instead of values, our semantic domain is therefore made of *symbolic* values defined in Figure 12. A symbolic value can be a CompCert value *val*. However, our semantics does not evaluate operators but instead constructs symbolic values which represent delayed computations. The exhaustive list of unary operators ($\texttt{op}_1$) and binary operators ($\texttt{op}_2$) is given in Figure 12. These are all the operators that are defined on CompCert values and needed to evaluate C programs. A symbolic value can also be an indeterminate value $\texttt{indet}(l)$ labelled by a location l. As we shall see in Section 6.3, indeterminate values will be used to model uninitialised memory. In the paper, we use a concise and concrete C-like syntax for symbolic values. For instance, we will write $(\texttt{ptr}(b,i) \mid \texttt{int}(3))\&\texttt{int}(3)$ for $\texttt{OpAnd}(\texttt{OpOr}(\texttt{ptr}(b,i), \texttt{int}(3)), \texttt{int}(3))$.

There are certain semantic operations which cannot operate on symbolic values. For instance, the `load` (resp. `store`) operation reads from (resp. writes to) a particular location. Similarly, for conditional statements, the condition needs to be evaluated to a boolean, i.e. $\texttt{int}(0)$ or $\texttt{int}(1)$ in order to execute to desired branch. We call *normalisation* the function which maps symbolic values to genuine values. Intuitively, a symbolic value *sv* normalises to a value v if *sv* always evaluates to v. In the following, we formalise these notions.

4.1 Evaluation of Symbolic Values

The rationale for introducing symbolic values is to overcome the limitations of the existing semantics of CompCert. As a result, evaluating a symbolic value using the existing semantics of pointers given in Figure 9 would not increase the expressive-

ness. Another approach would be to axiomatise further the semantics of pointers. For instance, we could state that the exclusive or ($\hat{}$) is idempotent and that 0 is a neutral element for bitwise or (|):

$$\begin{aligned} \texttt{ptr}(b,o)\,\hat{}\,\texttt{ptr}(b,o) &= \texttt{int}(0) \\ \texttt{ptr}(b,o) \mid \texttt{int}(0) &= \texttt{ptr}(b,o) \end{aligned}$$

We rule out this approach that is by essence partial.

As we intend to reason about the bit-level encoding of pointers, our evaluation of symbolic values returns a value v that represents this bit-level encoding, i.e. the evaluation does not return pointers. The evaluation of a symbolic value is therefore parameterised by:

- a mapping $cm : \texttt{block} \to \texttt{int}$ that associates to each block a concrete address, i.e. a 32-bit integer;
- and a mapping $im : \texttt{location} \to \texttt{byte}$ that associates to each indeterminate location a concrete byte value, i.e. a 8-bit integer.

In the rest of the paper, we call cm a concrete memory and im an indeterminate memory. Both cm and im bridge the gap between the high-level concepts of blocks and locations and a low-level memory represented as an array of bytes.

The evaluation is defined recursively (see Figure 12) on the structure of symbolic values. Evaluating a pointer value $\texttt{ptr}(b,i)$ results in the concrete address of this pointer in the concrete memory cm, that is $cm(b)+i$. Every other value evaluates to itself. Evaluating an indeterminate value $\texttt{indet}(l)$ results in the byte value that is stored at location l, i.e. $im(l)$. The evaluation of unary and binary operators consists in first evaluating recursively their operands, and then applying the appropriate operations (represented by the `eval_unop` and `eval_binop` functions, that map syntactic constructors to their semantics).

4.2 Well-formedness Condition for Concrete Memories

As stated earlier, a concrete memory cm maps blocks to a concrete address, representing the base pointer of this block. Definition 4 states the invariants that constrain this mapping for a given CompCert memory. To comply with the C standard and the Application Binary Interface, blocks cannot be allocated at arbitrary addresses but must satisfy alignment constraints. The alignment constraint depends on the number of bytes of the data structure and is given by the function `alignment_of_size`:

Definition 2 (`alignment_of_size`):

```
alignment_of_size size :=
  match size with
  |0|1      ⇒ 0
  |2|3      ⇒ 1
  |4|5|6|7  ⇒ 2
  | _       ⇒ 3
  end.
```

In particular, a byte has no alignment constraint; a 16-bit integer is 2^1-byte aligned; a 32-bit integer is 2^2-byte aligned and a 64-bit integer is 2^3-byte aligned. It follows that the alignment of a block is obtained by the function `alignment` which retrieves the size of a block and returns the number of trailing bits that are zeros in the concrete representation of the block.

Definition 3 `alignment(m,b) := let (lo,hi) := bound(m,b) in alignment_of_size(hi−lo)`.

Definition 4 (Valid concrete memory ): A concrete memory cm is valid for a memory m (written $cm \vdash m$), if and only if the three following properties are satisfied.

1. Valid locations lie in the range $]0; 2^{32} - 1[$.
   $\forall b\ o, \texttt{valid}(m, b, o) \rightarrow 0 < cm(b) + o < \texttt{Int.max\_unsigned}$
2. Valid locations from distinct blocks do not overlap.
   $\forall b\ b' o\ o', b \neq b' \rightarrow \texttt{valid}(m, b, o) \rightarrow \texttt{valid}(m, b', o') \rightarrow cm(b) + o \neq cm(b') + o'$
3. Blocks are mapped to suitably aligned addresses.
   $\forall b, cm(b)\ \texttt{mod}\ 2^{\texttt{alignment}(m,b)} = 0$

The first condition excludes 0 from the address space because it denotes the `NULL` pointer. It also excludes `Int.max_unsigned` $= 2^{32} - 1$ but for a more subtle reason that is due to pointers *one-past-the-end*. The C standard stipulates that, given an array of n elements, the addresses of successive elements (including n) are strictly increasing. Formally, we have: a+0 < a+1 ... a+(n−1)< a+n. Note that a+n is a pointer *one-past-the-end* of the array. We exclude $2^{32} - 1$ from the address space to prevent a possible wrap-around of a+n that would invalidate the inequality expected by the C standard. The second condition states the implicit property of a block-based memory model: valid addresses from distinct blocks do not overlap. The third condition makes sure that blocks are aligned according to their size.

4.3 Normalisation of Symbolic Values

To get an executable semantics, we require the normalisation primitive to be a function. The function, called `normalise` : `mem` $\rightarrow$ *sval* $\rightarrow$ *val*, takes as input a symbolic value sv and a memory m, and returns a value v. Ideally, we would like v to be such that sv evaluates to v for any indeterminate memory im and any concrete memory cm valid for m. Unfortunately, Example 2 shows that such a v does not always exist.

Example 2 Consider the symbolic values sv = `indet(`b, o`)` and sv' = `ptr(`$b, 0$`)` − `ptr(`$b', 0$`)`. For sv, there does not exist a value v such that $[\![sv]\!]^{im}_{cm} = v$ for every im. That would imply that $\forall im, im'.im(b, o) = im'(b, o)$, which is a contradiction. For sv', the difference of pointers evaluates to $cm(b) - cm(b')$ for every $cm \vdash m$. For different concrete memories, the evaluation returns different values.

When it is not possible to identify a unique value v, the normalisation returns the value `undef`. Definition 6 formalises the soundness criteria for a normalisation function. To capture the fact that it is always sound for a normalisation to return `undef`, it uses the relation $\leq$ (read *less defined than*) such that `undef` is less defined than any value. It is formally defined as these two rules:

Definition 5 ($\leq$ ): $\forall v, \texttt{undef} \leq v \qquad \forall v, v \leq v$

Definition 6 (Sound normalisation  **)**: Consider a memory m and a symbolic value sv. A value v is a sound normalisation of sv (written $\texttt{is\_norm}\ m\ sv\ v$) if the value v is *less defined than* the evaluation of sv for any valid concrete memory cm and any indeterminate memory im.

$$\forall cm \vdash m, \forall im, [\![v]\!]^{im}_{cm} \leq [\![sv]\!]^{im}_{cm}$$

A normalisation function `norm` is sound if for any memory m and symbolic value sv, we have that $\texttt{norm}\ m\ sv$ is a sound normalisation of sv.

For proving properties of the normalisation, for instance in Section 9.3, we use the more convenient formulation provided by Lemma 1 which is equivalent when the normalisation is not `undef`.

Lemma 1 (`is_norm_not_undef_spec` ): *For any memory m, symbolic value sv and value $v \neq \texttt{undef}$,*

$$\texttt{is\_norm}\ m\ sv\ v \Leftrightarrow \forall cm \vdash m, \forall im, [\![v]\!]^{im}_{cm} = [\![sv]\!]^{im}_{cm}.$$

Proof By definition of $\leq$. □

Example 3 Consider the code of Figure 5b. Unlike the existing semantics, operators are not strict in `undef` but construct symbolic values. Hence, in Line 7, we store in `bf.__bf1` the symbolic value sv defined by $(\texttt{indet}(l)\,\&\,\tilde{}\,\texttt{0x2})|(1{\ll}1\&\texttt{0x2})$. Next, we retrieve the value of bit-field `a1` and end up with the symbolic value $sv' = (sv{\ll}30) \gg 31$. Let us show that sv' normalises to $\texttt{int}(1)$, as expected (see Section 2.2.2).

We need to show that for any concrete memory cm and any indeterminate memory im, we have $[\![\texttt{int}(1)]\!]^{im}_{cm} \leq [\![sv']\!]^{im}_{cm}$ or equivalently, since $\texttt{int}(1) \neq \texttt{undef}$, that $[\![sv']\!]^{im}_{cm} = \texttt{int}(1)$.

$$\begin{aligned}
[\![sv']\!]^{im}_{cm} &= [\![(((\texttt{indet}(l)\,\&\,\tilde{}\,\texttt{0x2}) \mid (1{\ll}1\&\texttt{0x2})){\ll}30) \gg 31]\!]^{im}_{cm} \\
&= (([\![\texttt{indet}(l)\,\&\,\tilde{}\,\texttt{0x2}]\!]^{im}_{cm} \mid [\![1{\ll}1\&\texttt{0x2}]\!]^{im}_{cm}){\ll}30) \gg 31 \\
&= ((([\![\texttt{indet}(l)]\!]^{im}_{cm}\&\texttt{0xFFFFFFFD}) \mid \texttt{int}(2)){\ll}30) \gg 31 \\
&= (((im(l)\&\texttt{0xFFFFFFFD}) \mid \texttt{0x00000002}){\ll}30) \gg 31
\end{aligned}$$

Let us write the hexadecimal representation of $im(l)$ as $\texttt{0xNPQRSTUV[wxyz]}$ where letters `N` to `V` denote abstract hexadecimal digits and letters `w` to `z` inside square brackets denote abstract binary digits. Then, we can write the hexadecimal representation of sv' as:

$$\begin{aligned}
[\![sv']\!]^{im}_{cm} &= (((im(l)\&\texttt{0xFFFFFFFD}) \mid \texttt{0x00000002}){\ll}30) \gg 31 \\
&= (((\texttt{0xNPQRSTUV[wxyz]}\&\texttt{0xFFFFFFFD}) \mid \texttt{0x00000002}){\ll}30) \gg 31 \\
&= ((\texttt{0xNPQRSTUV[wx0z]} \mid \texttt{0x00000002}){\ll}30) \gg 31 \\
&= (\texttt{0xNPQRSTUV[wx1z]}{\ll}30) \gg 31 \\
&= \texttt{0x[1z00]0000000} \gg 31 \\
&= \texttt{0x00000001} = \texttt{int}(1)
\end{aligned}$$

Notice that the result of this computation is independent of the arbitrary hexadecimal and binary digits we chose for the value of $im(l)$, hence $\texttt{int}(1)$ is a sound normalisation of sv'.

According to Definition 6, `undef` is always a sound normalisation. However, this normalisation is of little interest. What we aim at is a normalisation that is defined as much as possible. This property is formalised by Definition 7.

Definition 7 (Complete normalisation): A normalisation function `norm` is complete if for all sound normalisations $\texttt{norm}'$, we have:

$$\texttt{norm}'(m, sv) \leq \texttt{norm}(m, sv)$$

4.4 Assessment of the Normalisation

As the normalisation function is a central component of the semantics, it belongs *de facto* to the Trusted Computing Base of the compiler. It is therefore crucial that it is an adequate model of reality. There is of course no formal way to establish such a result. Yet, we have informal indications that the specification of the normalisation is adequate.

A methodological argument is that the normalisation is defined with respect to a very concrete memory model where pointers are mapped to integers. This concrete model is our ground truth. From this point of view, the normalisation can be interpreted as an abstraction of concrete run-time values by a CompCert value. We also have a formal argument (see Section 8.2) that, in the absence of memory overflow, our model is a refinement of the model of CompCert and therefore both semantics coincide when the semantics of CompCert is defined. For the additional behaviours that we capture, we have an executable implementation of the normalisation (see Section 5) and experiments evaluation confirming that we get the right semantics (see Section 7).

In the following, we establish a key property of the normalisation, namely the existence of a unique most defined normalisation. It appears that such a property requires to tame the normalisation. To do that, we do not tamper with the definition of the normalisation but instead impose a decidable constraint, namely Property 1, on the memory allocation function which fails and stops the semantics early on if this well-formedness condition is not satisfied.

We believe that this restriction, though technical, is rather innocuous and actually only rules out possible normalisations that would be counter-intuitive as they would exploit *near out-of-memory* situations. In any case, it only restricts the expressiveness of the semantics but has no impact on programs for which the semantics is not stuck.

4.4.1 Taming the Normalisation: ruling out near-out-of-memory situations

It is straightforward to prove by contradiction that all the normalisation functions that are complete according to Definition 7 are equal. Yet, the uniqueness of the normalisation function does not ensure its existence. As shown by Example 4 there are corner cases for which the same symbolic value may have several equally defined sound normalisations.

Example 4 Suppose a memory m with a single block b of size $2^{32} - 9$. Because it is 8-byte aligned and the last address $(2^{32} - 1)$ is not in the range of valid addresses, the unique possible valid concrete memory cm for m is such that $cm(b) = 8$.

As a result, both values $\texttt{int}(8)$ and $\texttt{ptr}(b, 0)$ are a sound normalisation for the degenerate symbolic value $\texttt{ptr}(b, 0)$.

Intuitively, *near out-of-memory* situations are responsible for the fact that a single symbolic value could have several sound defined normalisations. For instance, the situation of Example 4 would not be possible if there were several concrete memories where the block b could be allocated at different addresses. To ensure the existence of a complete normalisation function, we generalise this idea and ensure that all the block-based memories m that we construct have Property 1.

Property 1 (Sliding Blocks): A memory m satisfies the Sliding Blocks property if for any block b, there exists at least two valid concrete memories cm and cm' that allocate b at different concrete addresses while allocating all the other blocks at the same address. Formally,

$$\forall\ b, \exists cm, cm', \bigwedge \begin{cases} cm \vdash m \\ cm' \vdash m \\ cm(b) \neq cm'(b) \\ \forall b' \neq b, cm(b') = cm'(b') \end{cases}$$

In Section 6.1, we prove that all the memories constructed by the allocation operation of the memory model satisfy Property 1.

4.4.2 Existence of a complete normalisation

Assuming Property 1, Theorem 2 states that a complete normalisation exists.

Theorem 2 (Existence of a complete normalisation): *There exists a complete normalisation function.*

Proof Consider a given memory m and a given symbolic value sv. The proof amounts to showing that a defined sound normalisation for sv is necessarily unique. In other words, given v and v' sound normalisations of sv such that $v \neq \texttt{undef}$ and $v' \neq \texttt{undef}$, we have to prove that $v = v'$.

By Lemma 1, because v and v' are sound normalisations, we get:

$$\forall im\ \ cm \vdash m, [\![v]\!]^{im}_{cm} = [\![sv]\!]^{im}_{cm} \wedge [\![v']\!]^{im}_{cm} = [\![sv]\!]^{im}_{cm}$$

By transitivity, we get Hypothesis 1:

$$\forall im\ \ cm \vdash m, [\![v]\!]^{im}_{cm} = [\![v']\!]^{im}_{cm} \tag{1}$$

The proof is by case analysis over v and v'.

- Case $v \neq \texttt{ptr}(b, o)$ and $v' \neq \texttt{ptr}(b', o')$. By Property 1, there exists $cm \vdash m$. Moreover, because v and v' do not contain pointers, their evaluation is independent from cm and we get from Hypothesis 1: $v = [\![v]\!]^{im}_{cm} = [\![v']\!]^{im}_{cm} = v'$. Hence, the property holds.
- Case $v = \texttt{ptr}(b, o)$. By Property 1, we exhibit $cm \vdash m$ and $cm' \vdash m$ such that Hypotheses 2 and 3 hold:

$$cm(b) \neq cm'(b) \tag{2}$$

$$\forall\ b' \neq b,\ \ cm(b') = cm'(b') \tag{3}$$

– Case $v' = \texttt{int}(i)$. From Hypothesis 1 using cm and cm', we get:

$$cm(b) + o = [\![v]\!]^{im}_{cm} = [\![v']\!]^{im}_{cm} = i = [\![v']\!]^{im}_{cm'} = [\![v]\!]^{im}_{cm'} = cm'(b) + o$$

As a result, $cm(b) = cm(b')$. This contradicts Hypothesis 2 and the property holds.

– Case $v' = \texttt{ptr}(b', o')$.
  • Case $b = b'$. By Hypothesis 1, we get:

$$cm(b) + o = [\![v]\!]^{im}_{cm} = [\![v']\!]^{im}_{cm} = cm(b) + o'$$

As a result, we deduce that $o = o'$ and the property holds.
  • Case $b \neq b'$. By Hypothesis 1, we get:

$$\begin{array}{l} cm(b) + o \;\; = [\![v]\!]^{im}_{cm} \;\; = [\![v']\!]^{im}_{cm} \;\; = cm(b') + o' \\ cm'(b) + o = [\![v]\!]^{im}_{cm'} = [\![v']\!]^{im}_{cm'} = cm'(b') + o' \end{array}$$

Because $cm(b') = cm'(b')$ (from Hypothesis 3), we get by transitivity that $cm(b) + o = cm'(b) + o$ and therefore $cm(b) = cm'(b)$. This contradicts Hypothesis 2 and the property holds.

– Other cases are symmetric and therefore the property holds. □

In the following, we write `normalise` for the unique sound and complete normalisation function.

5 Implementation of the Normalisation using an SMT Solver

As we mentioned in Section 3.2, CompCert ships with an executable interpreter for CompCert C, which is a valuable tool to test the semantics. In the previous section, we explained that the semantics of all the languages include normalisations in various places of the semantics, hence we need to provide an implementation for the normalisation, in order to get an executable interpreter. Given a memory m, there are finitely many valid concrete memories cm. It is thus decidable to compute a sound and complete normalisation and the naive algorithm consists in enumerating over all the valid concrete memories and checking that the symbolic values always evaluate to the same values. Yet, this naive approach is not practical.

Instead, we show that the normalisation can be encoded as a decision problem in the logic of bitvectors and efficiently solved by off-the-shelf SMT solvers. A bitvector of size n is the logic counterpart of a machine integer with n bits. This logic is therefore a perfect match for reasoning over machine integers. This decision problem will then be solved by a SMT (*Satisfiability Modulo Theory*) solver (e.g. Z3 [29], CVC4 [1]). A SMT solver takes as input a set of variables (bitvectors in our case) and constraints over these variables. Its purpose is to find a *model*. A model is a *valuation*, i.e. an assignment of actual values to variables, such that all the constraints are satisfied. Its output is either `unsat` (for *unsatisfiable*), meaning that there exists no valuation that satisfies the given problem, or $\texttt{sat}(M)$, meaning that M is a *model* of the input problem.

Our implementation using a SMT solver is not verified. Yet, this only limits the trust in the interpreter. Nevertheless, some trust is gained through testing that this implementation returns sound answers for our benchmarks. This implementation

is not part of the Trusted Computing Base of the compiler and therefore has no impact on its soundness: the semantics preservation proofs of the compiler passes are based on the specification of the normalisation, not on its implementation.

A normalisation operation `normalise` m sv consists of three parts: the memory state m, the symbolic value sv and the logic of the normalisation. We show how we translate each of these components into the language of the SMT.

5.1 Axiomatising the Memory

The first step in stating the normalisation problem as a SMT problem is to axiomatise the memory in the SMT language. To encode a memory m, we define one logical variable for each block in m. We then define two logical functions *size* and *alignment* mapping each block to its size and alignment respectively. The alignment is the number of trailing bits that must be zero. Next, we axiomatise the valid concrete memory relation by directly translating Definition 4 into first-order logic.

Example 5 Consider a memory m restricted to two blocks b_1 and b_2, with b_1 of bounds $[0, 4[$ (therefore its 2 trailing bits are zeros) and b_2 of bounds $[0, 8[$ (therefore its 3 trailing bits are zeros). The axiomatisation of m is given by the following formulae.

$$\begin{aligned}
\text{Block sizes:} \quad & size(b) = \begin{cases} 4 \text{ if } b = b_1 \\ 8 \text{ if } b = b_2 \\ 0 \text{ otherwise} \end{cases} \\
\text{Block alignments:} \quad & alignment(b) = \begin{cases} 2 \text{ if } b = b_1 \\ 3 \text{ if } b = b_2 \\ 0 \text{ otherwise} \end{cases} \\
\text{No overlap:} \quad & \forall b, b', o, o'. \bigwedge \begin{cases} b \neq b' \\ o < size(b) \quad \Rightarrow cm(b) + o \neq cm(b') + o' \\ o' < size(b') \end{cases} \\
\text{Address space:} \quad & \forall b, o. o < size(b) \Rightarrow 0 < cm(b) + o < \texttt{Int.max_unsigned} - 1 \\
\text{Alignment :} \quad & \forall b, cm(b) \bmod 2^{alignment(b)} = 0
\end{aligned}$$

5.2 Translating Symbolic Values into Logical Expressions

Next, we transform the symbolic value sv to be normalised into a logical symbolic value that we write sv^*. We replace pointers `ptr(`b, i`)` by the bitvector addition of the variable associated with block b and the bitvector representing the integer i. We replace different occurrences of `undef` by distinct fresh logical variables thus modelling that `undef` may take any value. Indeterminate values `indet(`l`)` are also modelled by fresh variables, though the same variable is used for every occurence of the same label, modelling the intuition of an arbitrary fixed value. Other values are mapped to their representation as bit-vectors. Unary and binary operations on symbolic values are mapped to their equivalent operations in terms of bitvectors.

5.3 Normalisation as SMT Queries

We now show how a SMT solver can be used to compute normalisations. As we will see, the queries are quite different depending on whether we expect the normalisation to result in a pointer or an integer value. However, we have seen (see Theorem 2) that it can result in either one or the other but never both. Hence, the implementation of the normalisation will take as a parameter whether we expect a pointer or an integer normalisation, and both will be tried.

Normalising into an integer. The algorithm to normalise sv^* into an integer is described in Algorithm 1. First, we generate the SMT query: $sv^* = i$, where i is a fresh logical variable. Suppose the formula is satisfiable for a value v for logical variable i. This means that there exists a valid concrete memory such that sv is evaluated into the value v. However, this value v is a sound normalisation only when it is the evaluation for *every* possible valid concrete memory. To ensure this, we generate a second SMT query: $sv^* = i \wedge i \neq v$. This query is expected to be unsatisfiable. It searches for an alternate valid concrete memory that would yield a different concrete value. If it is indeed unsatisfiable, then we return v as the normalisation of sv, because it means that *every* valid concrete memory yields this value v. On the other hand, if it is satisfiable, then there exists a different result with a different valid concrete memory, meaning that the result depends non-deterministically on the concrete memory. In this case the normalised value is `undef`.

Algorithm 1: Normalisation of sv into an integer

```
if SMT(sv* = i) = sat({i ↦ v}) then
  if SMT(sv* ≠ v) = unsat then
    return int(v)
  end if
end if
return undef
```

Example 6 Consider the memory m introduced in Example 5 and the symbolic value $sv = \texttt{ptr}(b_2, 0)\&\texttt{0x00000007}$. This symbolic value clear all bits but the three least significant from pointer $\texttt{ptr}(b_2, 0)$. It is expected to normalise to $\texttt{int}(0)$ because the last three bits must be 0, because of alignment constraints.

We generate the SMT query $b_2\&\texttt{0x00000007} = i$, where b_2 is a logical variable introduced to represent the block b_2. The SMT solver answers $i = 0$ because it has found a concrete memory cm where e.g. $cm(b_2) = 8$ and it has computed that $8\&7 = 0$. We now check that there is no other integer solution by submitting the following query: $b_2\&\texttt{0x00000007} = i \wedge i \neq 0$. The SMT solver answers `unsat`, indicating that no valid concrete memory yields an integer different from 0. Hence sv normalises into $\texttt{int}(0)$.

Consider now the symbolic value $sv = \texttt{ptr}(b_1, 0) < \texttt{ptr}(b_2, 0)$, with the same memory state m. The first SMT query $sv^* = i$ can be satisfied with e.g. $i = 0$, meaning that there is a valid concrete memory cm where b_1 is allocated after b_2, e.g. $cm(b_1) = 16$ and $cm(b_2) = 8$. We then submit the second SMT query:

$sv^* = i \wedge i \neq 0$. It is satisfied with $i = 1$ by a concrete memory where e.g. $cm(b_1) = 4$ and $cm(b_2) = 8$. Hence, sv normalises into `undef`.

Normalising into a pointer. Getting the normalisation of a pointer value is more complicated because there are several ways of decomposing an integer into a location made of a base and an offset. Theorem 2 tells us that only one such decomposition will be valid for all concrete memories. Algorithm 2 explains how we normalise symbolic values into pointers, and is based on the fact that a symbolic value sv can only have $\texttt{ptr}(b, o)$ as normalisation if b appears syntactically in sv. We first prove this fact, and then explain the algorithm.

Definition 8 (Syntactic appearance of blocks):

```
block_appears sv b :=
  match sv with
  | ptr(b', i)      ⇒ b = b'
  | op1 sv1         ⇒ block_appears sv1 b
  | sv1 op2 sv2     ⇒ block_appears sv1 b ∨ block_appears sv2 b
  | _               ⇒ ⊥
  end.
```

Lemma 2 (`norm_block_appears`): *For any memory m, for any symbolic value sv, if* `normalise` m $sv = \texttt{ptr}(b, i)$*, then the block b appears syntactically in sv.*

Proof The proof is by contradiction. Assume b does not appear in sv. Property 1 applied on block b provides two concrete memories cm and cm' such that

$$\begin{cases} cm \vdash m \\ cm' \vdash m \\ cm(b) \neq cm'(b) \\ \forall b' \neq b, cm(b') = cm'(b') \end{cases}$$

For any indeterminate memory im, we can derive the two following contradictory facts:

- Since b does not appear in sv, and cm and cm' agree on all blocks but b, we have $[\![sv]\!]^{im}_{cm} = [\![sv]\!]^{im}_{cm'}$.
- By Lemma 1, we have that $[\![sv]\!]^{im}_{cm} = cm(b) + i$ and $[\![sv]\!]^{im}_{cm'} = cm'(b) + i$. Since $cm(b) \neq cm'(b)$, we have that $[\![sv]\!]^{im}_{cm} \neq [\![sv]\!]^{im}_{cm'}$. □

As a result, given fresh logical variables b and o, we encode that b must be a block that appears in sv by asserting the logical constraint $b \in B$, where B is initially the set of blocks that appear in sv. We generate the SMT query $sv^* = b + o$. Suppose we get a model such that $b \mapsto b'$ and $o \mapsto o'$. Our next query checks whether there can be another pointer denoted by the same symbolic value in another valid concrete memory: $sv^* \neq b' + o'$. If the query is unsatisfiable, then the normalisation returns $\texttt{ptr}(b', o')$. Otherwise, if the query is still satisfiable, we know that $\texttt{ptr}(b', o')$ is not a sound normalisation of sv. We can therefore discard block b' from the candidates for the normalisation of sv (i.e. we remove it from the set B) and we iterate the search. This process eventually terminates because there are finitely many blocks b that appear syntactically in sv.

Algorithm 2: Normalisation of sv into a pointer

$B \leftarrow \{b \mid b \in sv\}$
while true do
  if $SMT(sv^* = b + o \wedge b \in B) = \textbf{sat}\{b \mapsto b', o \mapsto o'\}$ **then**
    if $SMT(sv^* \neq b' + o') = \textbf{unsat}$ **then**
      return $\texttt{ptr}(b', o')$
    else
      $B \leftarrow B \backslash \{b'\}$
    end if
  else
    return `undef`
  end if
end while
return `undef`

Example 7 Consider again the memory m of Example 5 and the symbolic value $sv = \texttt{ptr}(b_1, 1) - \texttt{ptr}(b_2, 2) + \texttt{ptr}(b_2, 4) + \texttt{indet}(b_3, 4) \,\&\, \texttt{int}(0x0)$. We process sv into a logical expression sv^* by replacing $\texttt{indet}(b_3, 4)$ by the fresh variable $x_{3,4}$:

$$sv^* = b_1 + 1 - b_2 - 2 + b_2 + 4 + x_{3,4} \,\&\, \texttt{0x0}$$

Notice that the two occurrences of b_2 cancel each other out, and that we have $\forall x, x \,\&\, \texttt{0x0} = 0$. The expression sv^* is therefore equivalent to $b_1 + 3$. This simplification is not actually made in the implementation and is merely present here for the sake of clarity.

The SMT query we need to solve is $b_1 + 3 = b + o$. The SMT solver may generate a valid concrete memory cm where $cm(b_1) = 4$ and $cm(b_2) = 8$, and propose the solution $b^* = b_2$ and $o^* = -1$, which satisfies the equation we gave as input. However, the query $sv^* \neq b^* + o^*$ is indeed satisfiable, for example with a concrete memory cm' identical to cm except that $cm(b_2) = 16$.

We begin the whole process again, with the extra constraint that $b \neq b_2$. A more natural solution is $b^* = b_1$ and $o^* = 3$. It turns out this is the only solution to this equation, as we can see by submitting this second query to the SMT solver, $b_1 + 3 \neq b_1 + 3$, which is obviously unsatisfiable. Therefore the symbolic value sv is normalised into the location $\texttt{ptr}(b_1, 3)$.

5.4 Relaxation and Optimisation of the SMT Encoding

The previous encoding of the memory is linear in the number of allocated blocks, as there is one definition for the *size* function and one for the *alignment* function for every block. Thus, as the memory gets bigger, the normalisation would get slower. In practice, we observe that the size of the memory has a dramatic (negative) impact on SMT solvers. To tackle the problem, we propose a relaxation of the SMT query that is independent of the number of allocated blocks and only depends on the size of the symbolic value to be normalised. A key observation is that a symbolic value can only be normalised if the corresponding SMT query has a unique solution. As a result, it is always sound to relax the SMT query and generate a weaker one (i.e. with potentially more solutions) provided the initial formula is satisfiable. Indeed, if there are more solutions, the normalisation will fail – this is always sound.

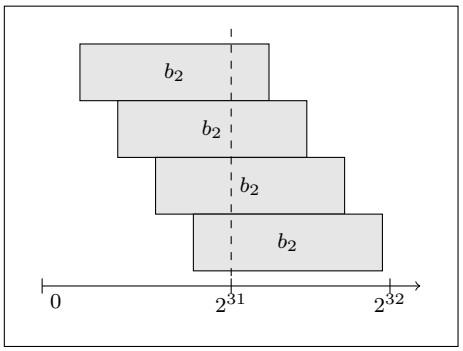


Fig. 13: Large blocks prevent some addresses from being allocated to others.

In our relaxation, we do not fully axiomatise the memory but only specify the bounds and alignments of the memory blocks B that appear syntactically in the symbolic value to be normalised. When normalising into a pointer, we also state explicitly in the SMT query that the normalisation, if it exists, should be a location (b, i) such that $b \in B$.

This relaxation is always sound, as we discussed before, for two reasons: 1) there always exists a valid concrete memory, thanks to our allocation algorithm; 2) we generate a weaker SMT query, with potentially more solutions. This relaxation is however not complete. It might miss a normalisation in pathological cases where blocks $b \in B$ are constrained not to appear at certain locations, because of other blocks $b' \notin B$. This is illustrated by Example 8.

Example 8 Consider a memory with 2 blocks b_1 of size 8 and b_2 of size 2^{31}. Figure 13 shows the possible addresses of block b_2. Because of size constraints, the concrete address 2^{31} will always be part of block b_2. Notice that block b_1 can be mapped in any of these concrete memories either before or after block b_2. However b_1 will never be at address 2^{31}. The symbolic value $sv = \texttt{ptr}(b_1, 0) == \texttt{int}(2^{31})$ therefore normalises into `false`.

Now if we relax the validity constraints on concrete memories to only account for blocks that appear syntactically in sv, then we will have some concrete memories where b_1 is at address 2^{31}. As a result, sv will evaluate to `true` in some valid concrete memories, and to `false` in some others. Because no unique value can be found, the normalisation will fail.

The normalisation of Example 8 requires a full axiomatisation of the memory and cannot be obtained using our relaxation. In our experience, we have never encountered pathological cases where the relaxation fails when a normalisation exists. In particular, it gives a defined normalisation to all the examples in this paper.

6 Implementation of the Memory Model

We propose a memory model that extends the model of CompCert with symbolic values capturing the result of otherwise undefined operations. In this section, we

Symbolic `memvals`:	`smemval` ::= `Symbolic`(sv, n)	n-th byte of symbolic value sv
Memory operations:	`palloc` $m\ lo\ hi = \lfloor (m', b) \rfloor$	Allocate a fresh block with bounds $[lo, hi[$. Fails if no concrete memory can be constructed.
	`free` $m\ b = \lfloor m' \rfloor$	Free (invalidate) the block b
	`load` $\kappa\ m\ b\ i = \lfloor sv \rfloor$	Read consecutive bytes (as determined by κ) at block b, offset i of memory state m. If successful, return the contents of these bytes as symbolic value sv.
	`store` $\kappa\ m\ b\ i\ sv = \lfloor m' \rfloor$	Store the symbolic value sv as one or several consecutive bytes (as determined by κ) at offset i of block b. If successful, return an updated memory state m'.

Fig. 14: The symbolic memory model

explain how to replace CompCert values by symbolic values in the memory model. Since concrete addresses are 32-bit machine integers, the address space is finite: we adapt the allocation operation to cope with this. We detail how to update the internal representation of symbolic values in memory. Next, we describe the handling of uninitialised values. Finally, we explain the changes required to the semantics of all of CompCert's intermediate languages, with a special focus on Clight's semantics. Figure 14 sums up the new memory model and will be referred to throughout this section.

6.1 Memory Allocation

In CompCert, memory allocation always succeeds and returns a new block of the requested size. This makes the implicit assumption that the memory is infinite. In our model, the semantics of the normalisation is based on a finite memory assumption where blocks are mapped to 32-bit addresses. As a result, our allocation function, `palloc`, is partial and may fail when there is not enough available memory (see Figure 14). Actually, `palloc` not only ensures the existence of a concrete memory but also provides a constructive proof that all the memories satisfy the stronger Property 1.

6.1.1 Allocation algorithm

The implementation of `palloc` is shown in Figure 15. Let us examine the code of the different functions. Compared to the existing `alloc` function, `palloc` takes an additional argument `al` which specifies the alignment of the block. To decide whether it is possible to allocate the block, `palloc` is guarded by the predicate `can_alloc`. If the predicate holds, the allocation succeeds and calls the existing CompCert allocation operation `alloc` and records the alignment. Otherwise, the allocation fails. The `can_alloc` predicate checks that the alignment is valid: it should be bigger than the alignment computed from the size but smaller than a maximum alignment `MA`.

For programs which do not explicitly perform dynamic memory allocation, the value of `MA` can be set to 3 where 3 is the maximum value of `alignment_of_size`. For other programs, `MA` would typically be the alignment of a kernel page (i.e. 12 for pages of 4kB). The `can_alloc` predicate also computes an address `addr` using the `fresh_addr` function. The concrete address `addr` is such that all the blocks can be allocated below `addr` at addresses that are $2^{\texttt{MA}}$-bytes aligned. The predicate `can_alloc` checks that there are still $2^{\texttt{MA}}$ reserved bytes above `addr`. As we shall see, this reserved space will be necessary to ensure Property 1. The fresh address is recursively computed by `alloc_blocks`. It allocates each block at the next available `MA`-bit aligned address and returns the next available address and the constructed concrete memory. It takes as argument two accumulators: `next_available` and `cur`. The accumulator `next_available` is the next available address and `cur` is the concrete memory currently being constructed. The `align` function has a quite complicated code: it is such that `align n amount` returns the smallest multiple of `amount` greater than or equal to `n`. The notation `cur[b ↦ x]` denotes a function that returns the same value as `cur` except for input `b` which is mapped to value `x`.

```
Definition align (n: Z) (amount: Z) :=
  ((n + amount − 1) / amount) ∗ amount.

Fixpoint alloc_blocks (bl : list (block ∗ Z)) (next_available: Z)
                      (cur : block → Z) : (Z ∗ (block → Z)) :=
  match bl with
  | nil ⇒ (next_available, cur)
  | (b, sz):: l ⇒ alloc_blocks l (align next_available 2^MA + sz)
                cur[b ↦ align next_available 2^MA]
  end.

Definition fresh_addr (bl : list (block ∗ Z)) : Z :=
 fst (alloc_blocks bl MA (λ b ⇒ 0))

Definition can_alloc (m: mem) (sz: Z) (al: Z) : bool :=
  let b := fresh_block m in
  let addr := fresh_addr ((b,sz)::blocks_of m) in
  alignment_of_size sz ≤ al ≤ MA && addr < Int.max_unsigned − 2^MA.

Definition palloc (m: mem) (sz: Z) (al: Z) : option (mem ∗ block) :=
  if can_alloc m sz al then ⌊ set_alignment (alloc m 0 sz) al ⌋ else ∅.
```

Fig. 15: Definition of the new allocation operation

6.1.2 Allocation Properties

Property 1 is a sufficient condition for the existence of a complete normalisation (see Section 4.3). It states that for any memory m, it is possible to rearrange the blocks so that there always exist two concrete memories which only differ on a single block. We show in Theorem 3 that this is a property of the allocation algorithm presented above.

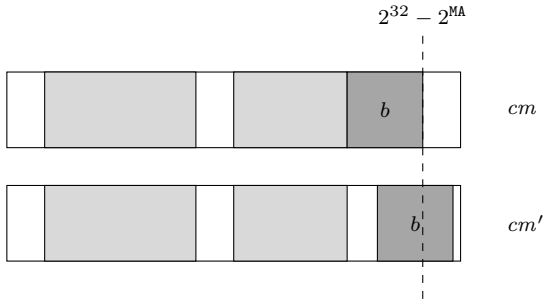


Fig. 16: Finding two different concrete memories for one block

Theorem 3 (Sliding Blocks): *A memory m is such that for any block b, there exist at least two valid concrete memories cm and cm' that allocate b at different concrete addresses while allocating all the other blocks at the same address. Formally,*

$$\forall\ b, \exists cm, cm', \bigwedge \begin{cases} cm \vdash m \wedge cm' \vdash m \\ cm(b) \neq cm'(b) \\ \forall b' \neq b, cm(b') = cm'(b') \end{cases}$$

Proof Any memory m is obtained from the initial memory m_0 by allocating new blocks using the allocation function `palloc`.

For the initial memory m_0, as there are no allocated blocks, all the concrete memories are valid. Given a block b, we can therefore construct cm and cm' such that $cm = (\lambda x.0)[b \mapsto 1]$ and $cm' = (\lambda x.0)[b \mapsto 2]$. Hence, the property holds.

Suppose that a memory m is obtained by the allocation function `palloc`. The algorithm checks that all the blocks fit in memory by running the function `fresh_addr` which constructs as witness a valid concrete memory cm and returns the first fresh address `addr`. The key insight of the proof is that the order of the blocks is not relevant for the success of `palloc`. Therefore, we can consider that, without loss of generality, any block b can be allocated last. The argument goes as follows. If the `alloc_blocks` function would follow a *first fit* allocation discipline, the alignment constraints may have an impact on the fragmentation of the witness concrete memory and therefore `palloc` may succeed or fail depending on the order the blocks are allocated. To prevent this, all the addresses computed by `alloc_blocks` are maximally aligned. Therefore, the success of the allocation is independent from the allocation order.

Moreover, the test `addr` $<$ `Int.max_unsigned` $- 2^{\mathtt{MA}}$ ensures that the last block, say b, can also be allocated at $cm(b) + 2^{\mathtt{MA}}$. This construction is depicted in Figure 16 where grey rectangles are allocated blocks and the darker block is b. White spaces in the figure represent free memory, i.e. memory that does not belong to any block. Hence the property holds for any block b. □

6.2 In-memory Data Representation

The memory content is no longer represented by the `memvals` that we described in Section 3. Instead, we use a generalized form called `smemval` (see Figure 14)

with a single constructor that subsumes all the existing ones and makes it possible to encode symbolic values. A `smemval` is merely a pair composed of a symbolic value *sv* and a natural number n denoting the n-th byte of the symbolic value *sv*, following the same principles as the `Pointer` constructor of `memval`. To get a uniform encoding, symbolic values of any type are mapped to an equivalent symbolic value of type `long`, i.e. a bitstring of 64 bits. The conversion is performed by the function `to_bits` : `memory_chunk` $\rightarrow$ *sval* $\rightarrow$ *sval* and the reverse conversion is performed by the function `from_bits` : `memory_chunk` $\rightarrow$ *sval* $\rightarrow$ *sval*. For example, symbolic values that represent floating-point values are converted to their binary representation using the `OpBitsofdouble` operator, introduced in Figure 12, hence `to_bits Mfloat64` *sv* is actually $\texttt{op}_1$ `OpBitsofdouble` *sv*. Likewise, the `from_bits` function interprets a bitstring as a symbolic value of the specified type. For example, `from_bits Mfloat64` *sv* interprets the bitstring *sv* as a floating-point expression: $\texttt{op}_1$ `OpDoubleofbits` *sv*.

Encoding a symbolic value *sv* into a list of `smemval`s according to a chunk κ is straightforward. It consists in building a list of $n =$ `size_chunk` κ elements of the form `Symbolic` (`to_bits` κ *sv*) i, $i \in 0, \ldots, n-1$. Decoding a list of `smemval`s into a symbolic value is somewhat more complicated. First, we show how to decode one `smemval`: `Symbolic` *sv* n. We define a function `extr` : *sval* $\rightarrow \mathbb{N} \rightarrow$ *sval* in Figure 17. It is defined recursively: the 0-th byte is obtained by masking the higher bits; the $(n+1)$-byte of *sv* is obtained by shifting *sv* 8 bits to the right, then taking the n-th byte of the resulting symbolic value.

```
Fixpoint extr (sv : sval) (n: nat) : sval :=
match n with
| O ⇒ sv & 0xFF
| S m ⇒ extr (sv >> 8) m
end.

Definition smv_to_sval (smv: smemval) : sval :=
match smv with Symbolic sv n ⇒ extr sv n
end.

Fixpoint concat (l : list smemval) : sval :=
match l with
| nil ⇒ 0
| a:: r ⇒ (smv_to_sval a) + (concat r) << 8
end.

Definition decode (l: list smemval) (κ : memory_chunk) : sval :=
  from_bits κ l.
```

Fig. 17: Decoding `smemval`s into symbolic values

Then we need to decode lists of such `smemval`s. This is done by converting each `smemval` into a symbolic value, and then concatenating those symbolic values: the `concat` function recovers the 64-bit bitvector that represents the original symbolic value, and the `decode` function applies the `from_bits` function to the result of `concat` with the appropriate chunk.

The operation `load` now loads symbolic values from memory and `store` now stores symbolic values into the memory, as Figure 14 shows.

6.3 Precise Handling of Uninitialised Values

Thanks to the $\texttt{indet}(b,i)$ construction offered by symbolic values, we are able to perform operations on (partially) uninitialised data (e.g. see the examples in Figure 4). We initialise the contents of freshly allocated blocks with $\texttt{indet}(b,i)$ where (b,i) denotes the location being initialised. This ensures the uniqueness of the identifiers.

Example 9 (Evaluation of symbolic values with uninitialized values): Let b be a block corresponding to a freshly allocated variable `x` of type `char`. The contents of the cell at location $(b,0)$ is initialized with $\texttt{indet}(b,0)$. The C expression `x - x` is first transformed into the symbolic value $\texttt{indet}(b,0) - \texttt{indet}(b,0)$. This symbolic value is later evaluated to $\texttt{int}(0)$, because for any im:

$$[\![\texttt{indet}(b,0) - \texttt{indet}(b,0)]\!]^{im}_{cm} = im(b,0) - im(b,0) = 0$$

6.4 Semantics of Clight

The modified Clight semantics of expressions is given in Figure 18. It is defined by judgements, parameterised by a global environment G (holding global variables and functions), a local environment E (holding local variables) and an initial memory state m. The evaluation of expressions is split between expressions in l-value and r-value positions. Expressions in l-value position evaluate to locations, whereas expressions in r-value position evaluate to symbolic values. The execution of a statement is also parameterised by a global environment G. A state is a tuple $\langle f, S, E, k, m\rangle$ where f is the function that we are in, S is the statement to be executed, E is a local environment (mapping local variables to their values), k is a continuation and m is a memory state. In the judgements, a, a_1, a_2 range over syntactic expressions and sv, sv_1, sv_2 range over symbolic values.

We only show the rules that introduce normalisations, and therefore differ from CompCert's Clight semantics. The evaluation of locations as r-values is described by the `deref_loc` predicate. It behaves differently depending on the type of the location being accessed (given by the `access_mode` function describing how a l-value of a given type must be accessed). If the expression has scalar type, then its value is loaded from memory at the location denoted by the expression. If the expression has array, function, structure or union type, its value is the location itself.

The `assign_loc` describes the behaviour of storing some symbolic value to a given location. If the type of the location is scalar, a memory store is performed and the resulting memory state is returned. However, if the type is a structure or a union, then it must be copied byte-wise (see rule (7)). The `load` and `store` operations expect a genuine location to read from or write to. Therefore, in the semantics of `deref_loc` and `assign_loc`, the symbolic values are normalised into genuine pointers before performing the memory operations.

Rules (9) and (10) deal with if-then-else statements. The condition is first evaluated to a symbolic value, which is then normalised into an integer i. Depending

Judgements

$$\begin{array}{ll} G, E \vdash a, m \Leftarrow \ell & \text{(evaluation of an expression in l-value position)} \\ G, E \vdash a, m \Rightarrow sv & \text{(evaluation of an expression in r-value position)} \\ G \vdash \langle f, S, E, k, m\rangle \to \langle f', S', E', k', m'\rangle & \text{(execution of a statement)} \end{array}$$

Dereference of a location:

$$\frac{\begin{array}{c}\texttt{access\_mode}(ty) = \texttt{By\_value}\ \kappa \\ \texttt{normalise}(m, sv_{ptr}) = \texttt{ptr}(b, i) \\ \texttt{load}\ \kappa\ m\ b\ i = \lfloor sv \rfloor\end{array}}{\texttt{deref\_loc}\ ty\ m\ sv_{ptr}\ sv} \quad (4) \qquad \frac{\texttt{access\_mode}(ty) \neq \texttt{By\_value}\ \kappa}{\texttt{deref\_loc}\ ty\ m\ sv_{ptr}\ sv_{ptr}} \quad (5)$$

Assignment to a location:

$$\frac{\begin{array}{c}\texttt{access\_mode}(ty) = \texttt{By\_value}\ \kappa \\ \texttt{normalise}(m, sv_{dst}) = \texttt{ptr}(b, i) \quad \texttt{store}\ \kappa\ m\ b\ i\ sv = \lfloor m' \rfloor\end{array}}{\texttt{assign\_loc}\ ty\ m\ sv_{dst}\ sv\ m'} \quad (6)$$

$$\frac{\begin{array}{c}\texttt{access\_mode}(ty) = \texttt{By\_copy} \\ \texttt{normalise}(m, sv_{src}) = \texttt{ptr}(b_{src}, i_{src}) \quad \texttt{loadbytes}\ m\ b_{src}\ i_{src}\ (\texttt{sizeof}(ty)) = \lfloor mvals \rfloor \\ \texttt{normalise}(m, sv_{dst}) = \texttt{ptr}(b_{dst}, i_{dst}) \quad \texttt{storebytes}\ m\ b_{dst}\ i_{dst}\ mvals = \lfloor m' \rfloor\end{array}}{\texttt{assign\_loc}\ ty\ m\ sv_{dst}\ sv_{src}\ m'} \quad (7)$$

Expressions in r-value position:

$$\frac{G, E \vdash a, m \Leftarrow sv_{ptr} \quad \texttt{deref\_loc}\ (\texttt{typeof}(a))\ m\ sv_{ptr}\ sv}{G, E \vdash a, m \Rightarrow sv} \quad (8)$$

Statements:

$$\frac{G, E \vdash a, m \Rightarrow sv \quad \texttt{normalise}(m, sv) = \mathrm{int}(i) \quad i \neq 0}{\langle f, \texttt{if}\ a\ \texttt{then}\ s_1\ \texttt{else}\ s_2, E, m, \dots\rangle \to \langle f, s_1, E, m, \dots\rangle} \quad (9)$$

$$\frac{G, E \vdash a, m \Rightarrow sv \quad \texttt{normalise}(m, sv) = \mathrm{int}(i) \quad i = 0}{\langle f, \texttt{if}\ a\ \texttt{then}\ s_1\ \texttt{else}\ s_2, E, m, \dots\rangle \to \langle f, s_2, E, m, \dots\rangle} \quad (10)$$

$$\frac{\begin{array}{c}G, E \vdash a_2, m \Rightarrow sv \quad \texttt{sem\_cast}(\texttt{typeof}(a_2), \texttt{typeof}(a_1), sv) = \lfloor sv' \rfloor \\ G, E \vdash a_1, m \Leftarrow sv_{ptr} \quad \texttt{assign\_loc}\ (\texttt{typeof}(a_1))\ m\ sv_{ptr}\ sv'\ m'\end{array}}{\langle f, a_1 = a_2, E, m, \dots\rangle \to \langle f, \texttt{Skip}, E, m', \dots\rangle} \quad (11)$$

Fig. 18: Semantics of Clight with symbolic values (excerpt)

on the value of i, the program will go in one branch or another. Rule (11) gives semantics to assignments. The semantics of statement `a1 = a2` is the following. First, evaluate a_2 into a symbolic value and cast it to the type of a_1, resulting in symbolic value sv'. Then, evaluate a_1 as a l-value, resulting in a symbolic value sv_{ptr} and then use the `assign_loc` predicate. The rest of the semantic rules is kept unmodified.

7 Experimental Evaluation

We have adapted the CompCert C interpreter so that we could test our semantics on real programs. The changes are similar to that described on Clight in Section 6.4. In order to be able to interpret real-world programs, we have stubs to model system calls such as `mmap` This system call is mapped to the `alloc` operation

of our memory model with appropriate parameters. Other system calls such as `open`, `read` or `write` that operate on files are mapped to a native implementation.

We have tested our C semantics with symbolic values on the CompCert benchmarks. Their size ranges from a few hundreds to a few thousands lines of code. We checked the absence of regression: when the CompCert interpreter returns a defined value, our interpreter returns exactly the same value.

We have also run our interpreter over Doug Lea's memory allocator [23] and on parts of the NaCl cryptographic library [3], which are challenging programs because they perform low-level pointer arithmetic; their size is about a few thousands lines of code. Our interpreter succeeds in giving semantics to memory management functions, such as `malloc`, `memalign` or `free`, built on top of `mmap`. As there is no other executable formal C semantics able to deal with low-level pointer arithmetic, we checked that the result of our interpreter was matching the output of `gcc`. Programs reading uninitialised variables have undefined semantics and `gcc` could exploit this to perform arbitrary computations. Yet, the output of `gcc` and our interpreter agree on examples similar to Figure 4. In the following, we detail some interesting patterns found in the benchmarks.

7.1 Pointer Arithmetic Using Alignment and Bitwise Operations

The implementation of `malloc` uses the macro `is_aligned` to check whether a pointer is aligned or not. The macro `is_aligned` performs a bitwise AND between a pointer `A` and a mask, here `0xF`, or $2^4 - 1$. It checks whether the pointer has its 4 least signifcant bits set to 0.

```
/* True if address A has acceptable alignment */
#define ALIGN_MASK 0xF
#define is_aligned(A) (((size_t)(A) & ALIGN_MASK) == 0)
```

Consider a block b allocated as the result of calling the `mmap` function. In our version of CompCert, we model a call to `mmap` by a `palloc` operation of the memory model with an alignment of 12 bits, modeling pages of 4kB with natural alignment.

Now, consider a pointer `p` modelled by the logical pointer $(b, 3)$. The macro `is_aligned` applied to `p` expands to $((\texttt{size\_t})(\texttt{p})\ \&\ \texttt{ALIGN\_MASK}) == 0)$. The symbolic value associated with this expression is $\texttt{ptr}(b, 3)\&\texttt{0xF} == 0$ and evaluates in a concrete memory cm into $(cm(b) + 3)\&\texttt{0xF} == 0$. Because of the alignment constraints on the block b, this symbolic value simplifies to $3 == 0$, which in turn evaluates to $\texttt{int}(0)$.

A similar example is the function `memalign(al,nb)`, where `al` must be a power of two (i.e. al$= 2^n$). The function dynamically allocates a `nb`-byte region, starting at an al-byte aligned address. When called with al $= 32$, the function computes conditions such as p&0x1F $== 0$ to check that the 5 last bits of `p` are zeros. The left-hand side of the comparison is evaluated in the same manner as in the example above, and the comparison is computed trivially.

7.2 Comparison Between Pointers and $(\texttt{void}*)(-1)$

As discussed in Section 2.1, several system calls, such as `mmap` or `sbrk`, are expected to return pointers but return $(\texttt{void}*)(-1)$ on error. Fig 3 shows an example of such

a call to `mmap`. Our normalisation gives a semantics to these comparisons between pointers and -1 using the following reasoning.

We know that pointers returned by `mmap` are aligned on a page boundary (2^{12} in our implementation, i.e. the 12 last bits of the pointer are zeros). When the allocation succeeds, the pointer can therefore never be -1 (in binary `0xFFFFFFFF`) thus allowing to evaluate this comparison between the pointer and -1.

7.3 Operations on Undefined Values

The example shown is Figure 4 is a simplified version of a C expression that appears in real-life programs. For example, the `memalign` function described in Section 7.1 features this kind of operations on undefined values.

The memory managed by the dynamic allocation functions is organised in memory chunks, which consist of two 32-bit words of meta-data followed by the memory chunk itself. The second word of meta-data stores the size of the chunk and two bits of other information. Initialising the meta-data is done with the C assignment $*p = (*p \ \& \ 0b1)|size|0b10$ (where the 0b prefix applies to constants in binary format). When the memory pointed by `p` is uninitialised, we construct the symbolic value `(indet(1) & 0b1)|size|0b10` . This symbolic value itself does not normalise, because its last bit is still indeterminate, however we are able to compute on this symbolic value, e.g. retrieve its second least significant bit with this symbolic value: $((indet(1) \ \& \ 0b1)|size|0b10) \ \& \ 0b10$. This normalises into `0b10`. This reasoning is made possible by the fact that `size` is a multiple of 4 (i.e. the last two trailing bits of `size` are zeros).

7.4 Copying Bytes between Memory Areas with `memmove`

Our semantics requires the target of jump instructions to be unique. This is a consequence of the fact that a symbolic value representing a conditional should normalise to some fixed boolean value. In other words, a program whose control-flow depends on the memory layout has an undefined behaviour. This dependance on the memory layout (e.g. on the memory allocator) is a portability bug that is detected by our semantics.

Indeed, in our experiments, we have encountered this situation for the `memmove` function (see Figure 19) which implements a memory copy even when the origin and destination memory regions do overlap. To cope with this situation, the `memmove` function performs the pointer comparison `dest <= src`. If the pointers `dest` and `src` point to distinct memory blocks, this comparison depends on the concrete memory and is therefore undefined for our memory model.

We have solved the issue by replacing the original condition `dest <= src` with the more involved condition $src <= dest \ \& \ dest < src + n$. This condition explicitly tests whether the memory regions overlap using the integer `n` which is the number of bytes to be copied. Notice that we use the bitwise & operator on purpose instead of the lazy boolean && operator. A && would force the evaluation of `src <= dest` which cannot be normalised. The new condition with a & constructs a symbolic value which is independent from the memory layout and has therefore always a defined normalisation. In particular, if the pointers are from distinct

```
void *memmove( void *s1, const void *s2, size_t n ) {
  char *dest = (char *) s1;
  const char *src = (const char *) s2;
  if ( dest <= src )
     while ( n-- ) { *dest++ = *src++; }
  else {
     src += n; dest += n;
     while ( n-- ) { *--dest = *--src; }
  }
  return s1;
}
```

Fig. 19: `memmove` with an undefined semantics

blocks, the condition is always false because locations from distinct blocks cannot overlap.

8 Properties of the Memory Model

The experiments show that our memory model gives a semantics to challenging low-level idioms. In this section, we study the formal properties of the model. We adapt and reprove the abstract interface of the CompCert memory model. Eventually, we prove that our new semantics of Clight simulates the original Clight semantics, thus cross-validating the models.

8.1 Good Variable Properties

CompCert's memory model exports an interface summarising all the properties of the memory operations necessary to prove the compiler passes. Those properties are called *good-variable properties* [26], and describe the behaviour of combinations of memory operations. For instance, the property `load_store_same` states that loading at an address that has just been written with some value v results in the same value v, converted to the appropriate chunk κ. The function `load_result` does this conversion. It consists of truncating integers to the expected size for chunks `Mint8xxx` and `Mint16xxx`. Formally, we have:

Theorem 4 (`load_store_same` ):
$\forall \kappa\ m\ b\ o\ v\ m', \texttt{store}\ \kappa\ m\ b\ o\ v = \lfloor m' \rfloor \rightarrow \texttt{load}\ \kappa\ m\ b\ o = \lfloor \texttt{load\_result}\ \kappa\ v \rfloor$.

Because we use symbolic values and delay their evaluation, this theorem does not hold anymore. This is illustrated by Example 10.

Example 10 Consider κ = `Mint16unsigned`, $o = 0$ and $v = \texttt{int}(3735928559) = \texttt{int}(\texttt{0xDEADBEEF})$. In CompCert, the `store` operation first encodes v into concrete bytes, keeping only the two least significant (because κ = `Mint16unsigned`) $b_1 = \texttt{0xBE}$ and $b_0 = \texttt{0xEF}$ and stores them at addresses $(b, 1)$ and $(b, 0)$ (respectively). The `load` then decodes these two bytes and computes the resulting value $v = \texttt{int}(256 * b_1 + b_2) = \texttt{int}(48879)$. Applying `load_result` with κ = `Mint16unsigned` to v results in the same integer $\texttt{int}(48879)$.

In our model however, the behaviour is slightly different. The `store` encodes each byte lazily, i.e. the addresses $(b, 1)$ and $(b, 0)$ do not contain concrete bytes but symbolic `smemval`s that denote them. Let sv be the symbolic value denoting the binary representation of value v for chunk `Mint16unsigned`, i.e. $sv =$ `to_bits Mint16unsigned` v. For example, the location $(b, 1)$ contains the `smemval` (`Symbolic` sv 1) that encodes byte number 1 of the binary representation of the original symbolic value v. The `load` first decodes `smemval`s into symbolic values, and then concatenates them to produce the final result. The `smemval` (`Symbolic` sv n) is decoded into $(sv \gg (8 * n))$ & `0xFF`. In our example we have $sv_1 = (sv \gg 8)$ & `0xFF` and $sv_2 = sv$ & `0xFF`. The concatenation is again expressed as a symbolic value based on shifts. The result of the `load` is then equal to the concatenation of sv_1 and sv_2, i.e. $L = (sv_1 \ll 8) + sv_2$. On the other hand, `load_result Mint16unsigned` v amounts to zeroing the 2 highest bytes, resulting in the symbolic value v & $(2^{16} - 1)$.

The theorem `load_store_same` clearly does not hold for Example 10: the two sides of the equation are different symbolic values. However, they are *equivalent*, i.e. they would always evaluate to the same value. This equivalence relation between symbolic values is formally defined as follows:

Definition 9 $sv_1 \equiv sv_2 \ := \ \forall \ cm \ im, [\![sv_1]\!]^{im}_{cm} = [\![sv_2]\!]^{im}_{cm}$.

We generalize `load_store_same` and every theorem of the memory model to use equivalence in lieu of syntactic equality when needed. We then state that there exists a symbolic value sv' that is the result of the `load` and this symbolic value is equivalent to the result we expect. The resulting theorems are of the form:

Theorem 5 (`load_store_same` **with symbolic values**):
$\forall \ \kappa \ m \ b \ o \ sv \ m', \texttt{store} \ \kappa \ m \ b \ o \ sv = \lfloor m' \rfloor \rightarrow$
$\exists sv', \texttt{load} \ \kappa \ m \ b \ o = \lfloor sv' \rfloor \wedge sv' \equiv \texttt{load\_result} \ \kappa \ sv.$

While the proof structure follows that of CompCert, the proof effort to port the whole memory model is substantial because we have to reason modulo equivalence of symbolic values.

8.2 Cross-validation of Memory Models

The semantics of the CompCert C language is part of the Trusted Computing Base of the compiler. Any modelling error can be responsible for a buggy, though formally verified, compiler. To detect a glitch in the semantics, a first approach consists in running tests and verifying that the CompCert C interpreter computes the expected value. With this respect, the CompCert C semantics successfully run hundreds of random test programs generated by CSmith [33]. Another indirect but original approach consists in relating formally different semantics for the same language. For instance, when designing the Clight semantics, several equivalences between alternate semantics were proved to validate this semantics [6]. Our memory model is a new and interesting opportunity to apply this methodology. In the following, we first describe the cross-validation of the Clight semantics that we performed, then we explain the errors that we discovered during the process of doing the proof.

8.2.1 Forward simulation between CompCert Clight and Symbolic Clight

In order to validate our semantics of Clight, we prove a forward simulation between CompCert Clight (CClight) and our modified symbolic Clight (SClight). That is, whenever a program has defined semantics in CClight, it will have the same semantics in SClight.

Of course, since the memory in SClight is finite and that in CClight is infinite, we cannot prove this simulation without further hypotheses. Consider for example a program that allocates a 2^{32}-byte array. Because the memory is infinite is CClight, this program has defined semantics and the simulation we are trying to prove requires that this program have defined semantics in SClight as well. This is however not possible because the array does not fit in our finite memory. Thus, we perform the proof under the hypothesis that memory allocation never fails. The theorem we prove is therefore the following: programs that have a defined semantics in CClight and do not exhaust the memory space have the same semantics in SClight.

To prove the simulation, we need to define an invariant `match_states` that relates CClight and SClight program states and that is preserved at every step of the semantics. This invariant is built on top of a relation `match_val` that relates values and symbolic values. We show in Example 11 a C program that we execute both with CClight and SClight semantics. We will then discuss our choice for the `match_val` relation.

Example 11 Consider the following C program: int i; return (&i != 0); It tests whether a valid pointer is different from `NULL`. We are interested in the return value of this program. We assume that variable `i` is allocated in block b. In CClight, the C expression is transformed into $\texttt{ptr}(b, 0)! = \texttt{int}(0)$, which in turn evaluates to `true`, i.e. $\texttt{int}(1)$. In SClight, we merely build the symbolic value $\texttt{ptr}(b, 0)! = \texttt{int}(0)$.

A natural candidate for `match_val v sv` is that sv must be the normalisation of v. However, this requires parameterizing `match_val` with a memory state and proving that all memory operations preserve `match_val`. As a matter of fact, the `free` operation does not preserve this normalisation-based `match_val`. For example, let m be the memory state of the program before returning its result. The symbolic value $\texttt{ptr}(b, 0)! = \texttt{int}(0)$ normalises to $\texttt{int}(1)$ in m. However, let m' be the memory state obtained after freeing block b from memory m, then the same symbolic value does not normalise in m' because $\texttt{ptr}(b, 0)$ is no longer valid. This is in accordance with the C standard[7] but a loss of completeness with respect with the existing CompCert semantics.

For the sake of the proof, we adapt the semantics of SClight to avoid this situation. The solution is to normalise symbolic values in a more eager manner i.e. before any write into memory or into a register, and only keep symbolic values when the normalisation fails. This is implemented by the function `simplify`:

Definition 10 $\texttt{simplify}\ m\ sv\ :=\ \texttt{let}\ v := \texttt{normalise}\ m\ sv\ \texttt{in}$
$\texttt{if}\ v = \texttt{undef}\ \texttt{then}\ sv\ \texttt{else}\ v.$

[7] [17][§6.2.4.2]: *"The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime."*

Back to our example, after introducing the simplifications, the `match_val` relation needs to relate `int`(1) and the simplification of `ptr`$(b, i)! =$ `int`(0), i.e. `int`(1). We define `match_val` as follows:

Definition 11 `match_val` $v\ sv\ :=\ \forall\ cm\ im,\ [\![v]\!]^{im}_{cm} \leq [\![sv]\!]^{im}_{cm}$.

We use $\leq$ instead of equality to account for the fact that SClight gives semantics to more programs than CClight, i.e. `undef` in CClight can be matched with any symbolic value in SClight.

A large part of the simulation proof is the preservation of C operators. That is, in a memory m, for any operation *op* that produces a value v in CClight, the same operation will produce an symbolic value sv, such that `match_val` v (`normalise` $m\ sv$). Indeed, if CClight produced a value $v \neq$ `undef`, then we must normalise it into the same value. This is stated formally in Lemma 3. The `sem_binop` function gives the CompCert semantics of a binary operator *op* applied to values v_1 of type t_1 and v_2 of type t_2. It is parameterised by a function `valid` m that takes a location (b, i) and returns *true* if and only if the location (b, i) is valid in memory m. This is needed for example for the semantics of pointer comparisons (see Figure 9). The function `sem_binop_sval` mimics the signature of `sem_binop` except that symbolic values replace values and it does not need information about the validity of pointers when constructing the symbolic values.

Lemma 3 (`expr_binop_preserved`):

$\forall\ op\ m\ v_1\ sv_1\ v_2\ sv_2\ t_1\ t_2\ v,$ `match_val` $v_1\ sv_1\ \rightarrow$ `match_val` $v_2\ sv_2 \rightarrow$
`sem_binop` $op\ v_1\ t_1\ v_2\ t_2$ (`valid` $m) = \lfloor v \rfloor \rightarrow$
$\exists sv,$ `sem_binop_sval` $op\ sv_1\ t_1\ sv_2\ t_2 = \lfloor sv \rfloor \wedge$ `match_val` v (`normalise` $m\ sv$).

8.2.2 An opportunity to discover bugs

During the proof, we have uncovered several issues, including silly mistakes in the evaluation of symbolic values: a particular cast operator was mapped to the wrong syntactic constructor. This is also during the proof that we have identified the issue of weakly valid pointers and therefore have excluded $2^{32} - 1$ from the address space (see Section 4.2). After these relatively easy fixes, we have found an interesting discrepancy with the semantics of CompCert C (version 2.4). The issue is related to the comparison with the `NULL` pointer. In CompCert, the `NULL` pointer is represented by the integer 0. The semantics therefore assumes that a location can never be equal to the `NULL` pointer. In our semantics, a location (b, i) can evaluate to 0 in case of wrap around. This is a glitch in the CompCert semantics that is illustrated by the code snippet of Figure 20. This program initialises a pointer `p` to the address of the variable `i`. In the loop, `p` is incremented until it equals `0` in which case the loop exits and the program returns `1`. With this program, the executable semantics of CompCert C returns `0` because `p==0` is always false whatever the value of `p`. However, when running the compiled program, the pointer is a mere integer, the integer eventually overflows; wraps around and becomes 0. Hence, the test holds and the program returns 1. We might wonder how the CompCert semantic preservation can hold in the presence of such a contradiction. Actually, the pointers are kept logical all the way through to the assembly level, and the comparison with the `NULL` pointer is treated the same during all the compilation process, thus even the assembly program in CompCert returns 0. The inconsistency

```
int main(){
  int i=0, *p = &i;
  for(i=0; i < INT_MAX; i++) {
    if (p++ == 0) {
      return 1;
    }
  }
  return 0;
}
```

Fig. 20: A `NULL` pointer comparison glitch

only appears when the assembly program is compiled into binary and run on a physical machine.

The fix consists in defining the semantics of the comparison with the `NULL` pointer only if the pointer is *weakly valid*. This causes the program to have undefined semantics at the C level as soon as we increment the pointer beyond its bounds. The issue has been acknowledged and is fixed since CompCert 2.5. After adjusting both memory models, we are able to prove that operators of CClight are preserved when transformed to SClight operators. Using this result, we prove a forward simulation between CClight and SClight, thus validating our formal semantics and that of CompCert.

9 Redesign of Memory Injections

As explained in Section 3.4, memory injections are an essential component for proving the correctness of the different compiler passes. In this section, we show how we adapt the definitions of memory injections for symbolic values. We also detail key properties of our injections with respect to the normalisation function.

9.1 Injection of Symbolic Values

The injection of values `val_inject` is lifted to symbolic values by the relation `sval_inj`. The injection f has the same type as before: when defined, it maps a block b to an offset δ in another block b'. A difference with the existing injection of values is that the injection function f is also used to inject indeterminate values. Rule (13) directly lifts the injection `val_inject` of values to symbolic values. Rules (14) and (15) propagate the injection by induction over the structure of symbolic values. Rule (12) states that `undef` can be injected into *any* symbolic value. This is a direct generalisation of the fact that the value `undef` can be injected into any value. Finally, rule (16) explains how to inject indeterminate values. It mimics rule (1) (see Section 3.4) that injects the location l of a pointer $\texttt{ptr}(b, i)$ using the injection function f. Injecting indeterminate values is needed to ensure that the locations are still fresh after an injection.

Note that the definition of `sval_inj` is syntactic. For example, we might have $\texttt{sval\_inj}\ f\ (\texttt{ptr}(b, i) + 1)\ (\texttt{ptr}(b', i + \delta) + 1)$, but not $\texttt{sval\_inj}\ f\ (\texttt{ptr}(b, i) + 1)\ (\texttt{ptr}(b', i + \delta + 1))$. As this is too restrictive, we consider the relation `sval_inject`

$$\frac{}{\texttt{sval\_inj}\ f\ \texttt{undef}\ sv} \quad (12) \qquad \frac{\texttt{val\_inject}\ f\ v_1\ v_2}{\texttt{sval\_inj}\ f\ v_1\ v_2} \quad (13)$$

$$\frac{\texttt{sval\_inj}\ f\ sv_1\ sv_2}{\texttt{sval\_inj}\ f\ \texttt{op}_1(sv_1)\ \texttt{op}(sv_2)} \quad (14) \qquad \frac{\texttt{sval\_inj}\ f\ sv_1\ sv_2 \quad \texttt{sval\_inj}\ f\ sv_1'\ sv_2'}{\texttt{sval\_inj}\ f\ \texttt{op}_2(sv_1, sv_1')\ \texttt{op}_2(sv_2, sv_2')} \quad (15)$$

$$\frac{f(b_1) = \lfloor (b_2, \delta) \rfloor}{\texttt{sval\_inj}\ f\ \texttt{indet}(b_1, i)\ \texttt{indet}(b_2, i+\delta)} \quad (16)$$

Fig. 21: Injection `sval_inj` of symbolic values

that is obtained by closing the relation `sval_inj` by the equivalence relation on symbolic values $\equiv$ (see Definition 9).

Definition 12 (`sval_inject`):
$\texttt{sval\_inject}\ f\ sv_1\ sv_2\ :=\ \exists\ sv_1'\ sv_2', sv_1 \equiv sv_1' \wedge \texttt{sval\_inj}\ f\ sv_1'\ sv_2' \wedge sv_2' \equiv sv_2.$

To define the injection of memories, we use the injection of symbolic values to inject memory values of the form `Symbolic` $sv\ n$. Two `smemval`s, mv_1 and mv_2 are in injection if the symbolic values they represent i.e. $\texttt{smv\_to\_sval}\ mv_1$ and $\texttt{smv\_to\_sval}\ mv_2$, are in injection (see Figure 17 for the definition of `smv_to_sval`).

Definition 13 (`memval_inject`):
$\texttt{memval\_inject}\ f\ mv_1\ mv_2 := \texttt{sval\_inject}\ f\ (\texttt{smv\_to\_sval}\ mv_1)\ (\texttt{smv\_to\_sval}\ mv_2).$

9.2 Injection of Memories

Given the previous generalisation to symbolic values, the definition of `mem_inject` is very similar to the original definition of CompCert. Definition 14 shows an excerpt from the `mem_inject` specification. The omitted part is inherited from CompCert and is not relevant to the discussion in this section.

Definition 14 (`mem_inject` ):
$\texttt{mem\_inject}\ f\ m_1\ m_2\ :\ \mathbb{P}\ :=\ \{$
$\quad \ldots$
$\texttt{mi\_align}: \quad \forall\ b\ b'\ \delta,\ f(b) = \lfloor (b', \delta) \rfloor \rightarrow$
$\qquad \texttt{alignment}(m_1, b) \leq \texttt{alignment}(m_2, b') \wedge 2^{[\texttt{alignment}(\texttt{m}_1, \texttt{b})]} \mid \delta;$
$\texttt{mi\_size\_mem}: \texttt{size\_mem}\ m_2 \leq \texttt{size\_mem}\ m_1$
$\}$

It features two distinctive properties, `mi_align` and `mi_size_mem`, that illustrate the main modifications due to symbolic values.

Absence of offset overflows. The existing specification of `mem_inject` has a property `mi_representable` which states that if $f(b) = \lfloor (b', \delta) \rfloor$, then for any valid offset o of b, the offset $o + \delta$ obtained after injection does not overflow (i.e. it is an integer that fits in 32 bits). With our memory model, this property can be derived from the other properties of the injection. Indeed, if o is a valid offset of b, then $o+\delta$ is a valid offset of b' (see the well-formedness properties of the injection in Section 3.4). Since $o + \delta$ is a valid offset of a block, then it is necessarily smaller than the size of the whole memory, which is itself, as we have explained in Section 6.1, strictly smaller than 2^{32}, therefore $o + \delta$ fits in a 32-bit integer.

Alignment constraints are modelled by the property `mi_align`. In CompCert, this is only a property of the offsets δ. As explained in Section 3.4, a chunk κ can be used to access a location (b, δ) if the offset δ is a multiple of `size_chunk` κ. The existing CompCert makes the implicit assumption that memory blocks are always sufficiently aligned to make the actual concrete address aligned as expected. In our model, blocks are given an explicit alignment. As a result, we can precisely state that an injection preserves alignement and is given by the `mi_align` property of Figure 14. We require that the target block is *at least as aligned as* the source block and that the offset δ is sufficiently aligned so that aligned locations are injected into *at least as aligned* locations.

The size constraint (`mi_size_mem`) is a property that is only present in our specification. It states that the memory after injection has to be *no larger* than the original memory. Here, the size is given by the function

$$\texttt{size\_mem}\ m\ :=\ \texttt{fresh\_addr}(\texttt{blocks\_of}\ m)$$

The function `fresh_addr` is presented in Figure 15; it ensures that all the blocks of the memory can fit below the computed fresh address. This restriction is needed to ensure that if a memory allocation succeeds for a source language, it also succeeds for the target language performing the allocation on an injected memory. This is illustrated by Theorem 6 given below. It states that provided that two memory states m_1 and m_2 are in injection, if we can allocate a block of size sz in m_1, then we can do the same in m_2 and the resulting memory states are in injection.

Theorem 6 (`palloc_parallel_inject` ):

$$\begin{array}{l}\forall\ f\ m_1\ m_2\ sz\ al\ m_1'\ b_1,\\ \quad 0 \leq sz \rightarrow \texttt{mem\_inject}\ f\ m_1\ m_2 \rightarrow \texttt{palloc}\ m_1\ sz\ al = \lfloor (m_1', b_1) \rfloor \rightarrow\\ \quad \exists\ m_2'\ b_2, \texttt{palloc}\ m_2\ sz\ al = \lfloor (m_2', b_2) \rfloor \wedge \texttt{mem\_inject}\ f[b_1 \mapsto \lfloor (b_2, 0) \rfloor]\ m_1'\ m_2'.\end{array}$$

Proof The insight of the proof is that the allocation `palloc` $m_1\ sz\ al$ succeeds for a memory m_1 that is at least as large as m_2. By definition of `palloc`, we have that

$$\texttt{size\_mem}\ m_1 + sz \leq \texttt{Int.max\_unsigned} - 2^{\texttt{MA}}$$

Moreover, by definition of the injection between m_1 and m_2, we also have that

$$\texttt{size\_mem}\ m_2 \leq \texttt{size\_mem}\ m_1$$

By arithmetics, it follows that $\texttt{size\_mem}\ m_2 + sz \leq \texttt{Int.max\_unsigned} - 2^{\texttt{MA}}$.

As a result, the allocation `palloc` $m_2\ sz\ al$ succeeds and returns a memory m_2' and a block b_2. It remains to prove that m_1' is in injection with m_2'. Though tedious, the proof of this part mimics the existing proof of CompCert. □

9.3 Preservation of Normalisation by Injection

This section details the proof of the main result relating normalisation and injection. The main theorem is the following:

Theorem 7 (`norm_inject` ):
$\forall\ f\ m\ m'\ sv\ sv', \texttt{all\_blocks\_injected}\ f\ m \rightarrow$
$\texttt{mem\_inject}\ f\ m\ m' \rightarrow \texttt{sval\_inject}\ f\ sv\ sv' \rightarrow$
$\texttt{val\_inject}\ f\ (\texttt{normalise}\ m\ sv)\ (\texttt{normalise}\ m'\ sv')$.

Informally, Theorem 7 states that the normalisation function preserves the injection of symbolic values. In particular, the result will be more defined (in the sense of the $\leq$ relation, see Definition 5) after injection. The intuition is that a memory injection amounts to merging blocks. As a result, pointer arithmetics gets more defined and therefore more symbolic values get a defined normalisation.

To formally prove this result, it is necessary to introduce the counterpart of memory injections for concrete memories and indeterminate memories. The definitions `inj_cm` and `inj_im` are similar; `inj_cm` states that blocks that are in injection are mapped to the same concrete address in both concrete memories and `inj_im` states that locations that are in injections have the same indeterminate value.

Definition 15 (`inj_cm` **and** `inj_im` ):
$\texttt{inj\_cm}\ f\ cm\ cm' := \forall\ b\ b'\ \delta,\ f(b) = \lfloor (b', \delta) \rfloor \rightarrow cm(b) = cm'(b') + \delta.$
$\texttt{inj\_im}\ f\ im\ im' := \forall\ l\ l',\ \texttt{sval\_inj}\ f\ \texttt{indet(}l\texttt{)}\ \texttt{indet(}l'\texttt{)} \rightarrow im(l) = im'(l').$

Given these definitions, we now prove the following result about the evaluation of symbolic values in injections in such concrete memories and indeterminate memories.

Lemma 4 (`eval_sval_inject` ):
$\forall\ f\ cm\ cm\ im\ im'\ sv\ sv',\ \texttt{inj\_cm}\ f\ cm\ cm' \rightarrow \texttt{inj\_im}\ f\ im\ im' \rightarrow$
$\texttt{sval\_inject}\ f\ sv\ sv' \rightarrow [\![sv]\!]^{im}_{cm} \leq [\![sv']\!]^{im'}_{cm'}$.

Proof By definition of `sval_inject`, there exists sv_1 and sv_2 such that $sv \equiv sv_1$ and $sv' \equiv sv_2$ and $\texttt{sval\_inj}\ f\ sv_1\ sv_2$. After rewriting the equivalences, the proof amounts to showing

$$[\![sv_1]\!]^{im}_{cm} \leq [\![sv_2]\!]^{im'}_{cm'}.$$

The proof is by induction over $\texttt{sval\_inj}\ f\ sv_1\ sv_2$.

- Case $sv_1 = \texttt{undef}$. Then, $[\![sv_1]\!]^{im}_{cm} = \texttt{undef}$. Because $\forall v, \texttt{undef} \leq v$, the property holds.
- Case $sv_1 = v$ and $sv_2 = v'$ where v and v' are values such that $\texttt{val\_inject}\ f\ v\ v'$. We must prove that $[\![v]\!]^{im}_{cm} \leq [\![v']\!]^{im'}_{cm'}$. The proof is by case analysis over $\texttt{val\_inject}\ f\ v\ v'$.
  - $v = \texttt{undef}$. Then $[\![v]\!]^{im}_{cm} = \texttt{undef}$ and the property holds.
  - $v = \texttt{ptr(}b, i\texttt{)}$ and $v' = \texttt{ptr(}b', i + \delta\texttt{)}$ and $f(b) = \lfloor (b', \delta) \rfloor$. On one hand we have $[\![v]\!]^{im}_{cm} = cm(b) + i$, and on the other hand, we have $[\![v']\!]^{im'}_{cm'} = cm'(b') + (i + \delta)$. Because cm and cm' are in injection, we know that $cm(b) = cm'(b') + \delta$ and the property holds.
  - v and v' are neither `undef` nor pointers. In this case $v = v'$, and their evaluations do not depend on the concrete memory and are therefore equal.
- Case $sv_1 = \texttt{indet(}b, i\texttt{)}$ and $sv_2 = \texttt{indet(}b', i + \delta\texttt{)}$ and $f(b) = \lfloor (b', \delta) \rfloor$. This case is similar to the proof of the case of pointers above.

– Case $sv_1 = \texttt{op}_1\ sv_1'$ and $sv_2 = \texttt{op}_1\ sv_2'$ and $\texttt{sval\_inj}\ f\ sv_1'\ sv_2'$. The induction hypothesis gives us:

$$\forall cm \vdash m, \forall im, [\![sv_1']\!]^{im}_{cm} \leq [\![sv_2']\!]^{im'}_{cm'}$$

We have on one hand $[\![sv_1]\!]^{im}_{cm} = [\![\texttt{op}_1\ sv_1']\!]^{im}_{cm} = \texttt{eval\_unop}(\texttt{op}_1, [\![sv_1']\!]^{im}_{cm})$ and on the other hand $[\![sv_2]\!]^{im'}_{cm'} = [\![\texttt{op}_1\ sv_2']\!]^{im'}_{cm'} = \texttt{eval\_unop}(\texttt{op}_1, [\![sv_2']\!]^{im}_{cm})$. The property holds because `eval_unop` is a morphism for $\leq$.

– Case $sv_1 = sv_3\ \texttt{op}_2\ sv_4$ and $sv_2 = sv_3'\ \texttt{op}_2\ sv_4'$. The property holds by induction hypothesis using the same arguments as for the unary operators. □

The following lemma is an important step in the proof of the `norm_inject` theorem. It claims that if a symbolic value sv can be injected by f, then its normalisation can also be injected.

Lemma 5 (`sval_inject_val_inject` 🦕):
$\forall\ f\ m\ sv\ sv'\ v,\ \texttt{sval\_inject}\ f\ sv\ sv' \rightarrow$
$\texttt{normalise}\ m\ sv = v \rightarrow \exists v', \texttt{val\_inject}\ f\ v\ v'.$

Proof By definition of `sval_inject` we have for some sv_1 and sv_2

$$sv \equiv sv_1 \wedge \texttt{sval\_inj}\ f\ sv_1\ sv_2 \wedge sv_2 \equiv sv'.$$

Since the normalisation is invariant under $\equiv$, we have $\texttt{normalise}\ m\ sv_1 = v$ and it remains to prove:

$$\texttt{sval\_inj}\ f\ sv_1\ sv_2 \rightarrow \texttt{normalise}\ m\ sv_1 = v \rightarrow \exists v', \texttt{val\_inject}\ f\ v\ v'.$$

The proof is by case analysis over v.

– Case $v = \texttt{undef}$. By rule (2) of `val_inject` ($\forall v, \texttt{val\_inject}\ f\ \texttt{undef}\ v$), the property holds.
– Case $v \neq \texttt{ptr}(b, i)$. By rule (3) of `val_inject` ($\forall v, \texttt{val\_inject}\ f\ v\ v$), the property holds.
– Case $v = \texttt{ptr}(b, i)$. From Lemma 2 (see Section 4.4.1), we have that b appears syntactically in sv_1. By direct induction over $\texttt{sval\_inj}\ sv_1\ sv_2$, it follows that $f(b) = \lfloor (b', \delta) \rfloor$ for some b' and δ. By rule (1) of `val_inject`, $v' = \texttt{ptr}(b', i + \delta)$ is in injection with v and the property holds. □

The previous lemmas play a major role in the proof of Theorem 7 whose statement is recalled below. As precondition, the theorem requires that all the blocks with a positive size are injected i.e. the injection function f is defined for all the allocated blocks.

Definition 16 $\texttt{all\_blocks\_injected}\ f\ m : \mathbb{P} :=\ \forall\ b,\ \texttt{size}\ m\ b > 0 \rightarrow f(b) \neq \emptyset.$

As we show in the proof, this condition is needed to ensure that it is always possible to construct concrete memories and indeterminate memories that are also in injection. We repeat the main theorem below and prove it.

Theorem 7 (`norm_inject` 🦕):
$\forall\ f\ m\ m'\ sv\ sv', \texttt{all\_blocks\_injected}\ f\ m \rightarrow$
$\texttt{mem\_inject}\ f\ m\ m' \rightarrow \texttt{sval\_inject}\ f\ sv\ sv' \rightarrow$
$\texttt{val\_inject}\ f\ (\texttt{normalise}\ m\ sv)\ (\texttt{normalise}\ m'\ sv').$

Proof The proof is by case analysis over the result, say v, of the normalisation `normalise` m sv.

- Case $v =$ `undef`. As rule (12) states that `undef` can be injected to any value, the property holds.
- Case $v \neq$ `undef`. From Lemma 5, we can always construct a value v' such that `val_inject` f v v'. To prove the property, it suffices to show that v' is indeed the result of the normalisation of sv', i.e. we have to prove the following:

$$\forall\ cm' \vdash m', \forall im', [\![v']\!]^{im'}_{cm'} \leq [\![sv']\!]^{im'}_{cm'}.$$

To relate the evaluations before and after injection, we exhibit a concrete memory cm and an indeterminate memory im defined by:

$cm(b) =$ `match` $f(b)$ `with`$\mid \lfloor (b', \delta) \rfloor \Rightarrow cm'(b') + \delta \mid \emptyset \Rightarrow 0$ `end`.
$im(b, i) =$ `match` $f(b)$ `with` $\mid \lfloor (b', \delta) \rfloor \Rightarrow im'(b', i + \delta) \mid \emptyset \Rightarrow 0$ `end`.

This construction inverts the injection. It computes a concrete memory cm such that $cm \vdash m$ and `inj_cm` f cm cm' and an indeterminate memory im such that `inj_im` f im im'. Note that this construction is only valid when all blocks are injected since non-injected blocks are mapped to an invalid concrete address (0). The intuition for this construction is given by Fig 22. On the left, Figure 22 depicts the compatible concrete memories before injection. For this example, the injection has the effect of concatenating the blocks b_1 and b_2, i.e. $f(b_1) = \lfloor (b, 0) \rfloor$ and $f(b_2) = \lfloor (b, \delta) \rfloor$ where δ is the upper bound of b_1. The left arrows $\leftarrow$ show the effect of the function `inj_cm` which inverts the injection. The construction of `inj_im` is similar.
By Lemma 1 and because $cm \vdash m$, we get Hypothesis 4:

$$\forall\ im, [\![v]\!]^{im}_{cm} = [\![sv]\!]^{im}_{cm}. \tag{4}$$

By applying Lemma 4 for v and v' and again for sv and sv', we also get:

$$[\![v]\!]^{im}_{cm} \leq [\![v']\!]^{im'}_{cm'} \qquad [\![sv]\!]^{im}_{cm} \leq [\![sv']\!]^{im'}_{cm'}$$

Because $v \neq$ `undef`, we have $[\![v]\!]^{im}_{cm} = [\![v']\!]^{im'}_{cm'}$, and by transitivity, we get:

$$[\![v']\!]^{im'}_{cm'} = [\![v]\!]^{im}_{cm} = [\![sv]\!]^{im}_{cm} \leq [\![sv']\!]^{im'}_{cm'}$$

As a result, the property holds. □

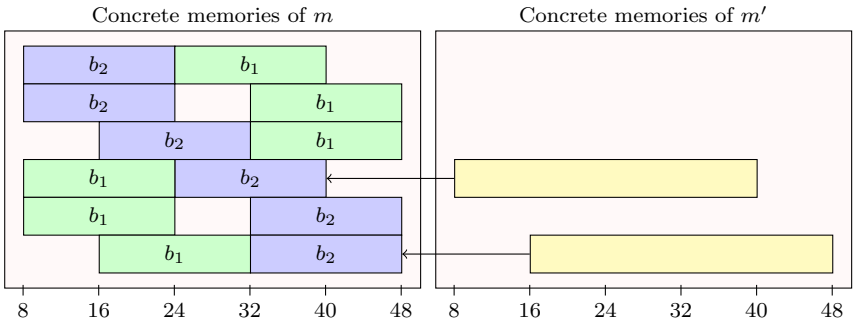


Fig. 22: Injection of concrete memories

10 Proof of the Front-end of the CompCert Compiler

This section gives a high-level account of the proof of the front-end of CompCert using symbolic values. The overall structure of the front-end is depicted in Figure 7. It is composed of four languages and three transformations. Details about the different languages and transformations can be found in Section 3.2. In the following, we first present how the semantics are modified to accommodate for symbolic values. For each transformation, we also highlight the difficulty of the proof with respect to our memory model.

10.1 Semantics of the Front-end Languages with Symbolic Values

The semantics of all intermediate languages need to be modified in order to account for symbolic values. In principle, the transformation consists in replacing values by symbolic values *everywhere* and introducing the normalisation function when necessary. In reality, the transformation can be more subtle because, for instance, certain intermediate semantic functions explicitly require locations represented as pairs (b, δ). In such situations, a naive solution consists in introducing a normalisation. Sometimes, the added normalisations are spurious and break the semantics preservation proofs when subsequent semantics do not have a matching normalisation. The right approach consists in delaying normalisations as much as possible. Normalisations are therefore introduced before memory accesses, in a seamless way. Indeed, the memory model comes with high-level operations `loadv` and `storev` which now take a symbolic value instead of a block and offset. These operations include a normalisation to recover a block and an offset, and then perform the `load`/`store` operation. Normalisations are also introduced when evaluating the condition of `if` statements. For CompCert C[8], normalisations are also added to model the lazy evaluation of && and || operators. Using this strategy we have adapted the semantics of the 4 languages of the front-end.

10.2 From CompCert C to Clight

The main purpose of the transformation from CompCert C to Clight is to pull side-effects out of expressions. The original proof is subtle and required a significant proof effort from CompCert's authors. One reason is that this is the only compiler pass which requires an explicit backward simulation proof due to the non-deterministic nature of the semantics of CompCert C.

However, this transformation preserves all the memory accesses. As a result, the simulation relation stipulates that the memories are syntactically the same for both the source and target programs. It follows that the existing proof can be reused almost unchanged providing that the normalisations are introduced at the right place. As hinted above, CompCert C includes many constructs that require a normalisation but are compiled away in Clight.

[8] These constructs are absent from other languages.

10.3 From Clight to C♯minor

The compilation from Clight to C♯minor translates loops and `switch` statements into simpler control structures. This pass also performs type-directed transformations and removes redundant casts. For example, it translates the expression $\texttt{p} + 1$ with `p` of type $\texttt{int} *$ into the expression $\texttt{p} + \texttt{sizeof(int)}$. For the existing memory model, both expressions compute exactly the same value. However, with symbolic values, syntactic equality is too strong a requirement. The simulation proof requires a weaker equivalence relation. A natural candidate is the equality of the normalisation. However, this relation is too weak and fails to pass the induction step. Indeed, when expressions e_1 and e_2 have the same normalisation ($sv_1 \equiv_N sv_2$), it is not always the case that $\texttt{op}_1\ sv_1 \equiv_N \texttt{op}_1\ sv_2$. Take for example $sv_1 = \texttt{indet}(b, i)|\texttt{0x1}$ and $sv_2 = \texttt{indet}(b, i)\&\texttt{0x0}$. The normalisations of sv_1 and sv_2 are `undef`, hence equal to each other. However, imagine the operation `lastbit` which retrieves the last bit of an integer. Now $\texttt{lastbit}(sv_1)$ normalises into $\texttt{int}(1)$ and $\texttt{lastbit}(sv_2)$ normalises into $\texttt{int}(0)$.

A stronger relation is the equivalence of symbolic values, introduced in Section 8.1. This relation is lifted to `smemval`s and memories:

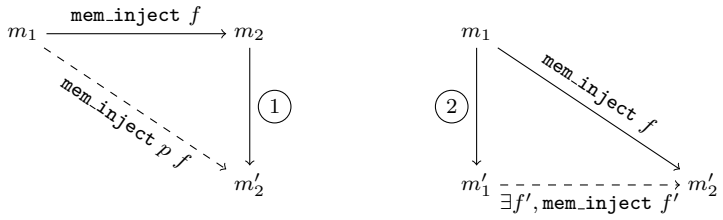
Definition 17 $\texttt{smemval\_eq}\ mv_1\ mv_2\ :=\ \texttt{smv\_to\_sval}\ mv_1 \equiv \texttt{smv\_to\_sval}\ mv_2$.

Two memory states m_1 and m_2 are equivalent, written $m_1 \equiv_m m_2$, if for all the locations both memories hold equivalent `smemval`s according to `smemval_eq`. To carry out the proof, we also extend the interface of the memory model and prove that the memory operations are morphisms for the equivalence relation. For example, we prove that starting from memory states m_1 and m_2 such that $m_1 \equiv_m m_2$, storing equivalent symbolic values at the same addresses will result in memory states m_1' and m_2' that are equivalent, i.e. $m_1' \equiv_m m_2'$. With these modifications, the compiler pass can be proved semantics preserving using the existing proof structure.

10.4 From C♯minor to Cminor

From the proof point of view, the compiler pass from C♯minor to Cminor is the most challenging. The reason is that this particular pass is responsible for allocating the stack frame. Therefore, it transforms significantly the memory layout and therefore the memory accesses. After the transformation, the stack frame is a single block and local variables are accessed via offsets within this block. The proof introduces a memory injection stating how the blocks representing local variables in C♯minor are mapped into the single block representing the stack frame in Cminor.

The existing proof can be adapted with our generalised notion of injection (see Section 9) with the notable exceptions of two intermediate lemmas whose proofs need to be completely re-engineered. The problem is related to the preservation of the memory injection when allocating and de-allocating the variables in C♯minor and the stack frame in Cminor. The structure of the original proof is depicted in Figure 23 where plain arrows represent hypotheses and the dotted arrow the conclusion. The existing proof first allocates the stack frame in memory m_2 to obtain the memory m_2'. It then establishes that the existing injection between the initial memories m_1 and m_2 still holds with the memory m_2'. In a second step,



1. allocation of stack frame
2. allocation of local variables

Fig. 23: Structure of `match_callstack_alloc_variables`'s proof in CompCert

the memory m'_1 is obtained by allocating variables in memory m_1 and the proof constructs an injection thus concluding the proof.

With our memory model, memory injections need to reduce the memory usage – this is needed to ensure that allocations cannot fail. Here, this is obviously not the case because the memory m'_2 contains a stack frame whereas the corresponding variables are not yet allocated in m_1.

Our modified proof is directly by induction over the number of allocated variables. In this case, we prove that if the variables do fit into memory, then so does the stack frame. Note that to accommodate for alignment and padding the stack frame might allocate more bytes than the size of the variables themselves. For example, consider a variable `x` of type `char` and a variable `y` of type `int`. The size of the variables is 5 bytes, but the stack frame needs to be 8-byte-wide because the integer has alignment constraints. However, remember that our allocation algorithm makes a worst-case assumption about alignment therefore there is always enough space to allocate the stack frame. Indeed, in the previous example, each of the variables `x` and `y` took 8 bytes in memory, therefore the stack frame takes less space. We therefore conclude that the memories m'_1 and m'_2 are in injection.

At function exit, the variables and the stack frame are freed from memory. As before, the arguments of the original proof do not hold with our memory model and we adapt the two-step proof with a direct induction over the number of variables. To carry out this proof and establish an injection we have to reason about the relative sizes of the memories.

Consider memory states m_1 and m_2 where `mem_inject` f m_1 m_2 for some f. We are about to de-allocate variables (of size *sz_vars*) from m_1 and the stack block (of size *stack_size*) from m_2. We know that *stack_size* $\leq$ *sz_vars* and also that `size_mem` $m_2 \leq$ `size_mem` m_1. We need to show that the sizes of the memory after de-allocation satisfy the `mi_size_mem` constraint (see section 9 for details), i.e. that `size_mem` $m_2 -$ *stack_size* $\leq$ `size_mem` $m_1 -$ *sz_vars*.

To prove this, we need more information. Our solution consists in enriching our invariant with a property of the sizes of the memories at every function entry. Instead of recording the relative sizes of the memory states only at the current program point, we record the whole history of related memory states in a predicate `size_mem_preserved` that we define. In particular, we know that at every point in the program, the memory in the source is always at least as large as the memory in the target. The proof regarding the sizes of the memory states after freeing the

```
Inductive size_mem_preserved:
  list mem → list mem → callstack → Prop :=
| smo_nil: ∀ m1 m2,
  size_mem m2 ≤ size_mem m1 →
  size_mem_preserved (m1::nil) (m2::nil) nil
| smo_cons: ∀ m1 m2 m1' lm1 m2' lm2 f cs,
  size_mem m1' = size_mem m1 − sz_vars f →
  size_mem m2' = size_mem m2 − stack_size f →
  size_mem m2 ≤ size_mem m1 →
  size_mem_preserved (m1' :: lm1) (m2' :: lm2) cs →
  size_mem_preserved (m1 :: m1' :: lm1)
                     (m2 :: m2' :: lm2) (f :: cs).
```

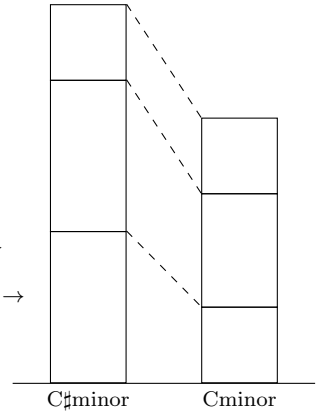


Fig. 24: The `size_mem_preserved` predicate

variables and the stack frame is then simple: we just need to pick that proof from our history.

Figure 24 gives the definition of `size_mem_preserved` on the left and a visual representation of the sizes of related memories on the right. We use the existing notion of call stack (`callstack`), which is a list of functions frames. A frame is a proof object relating C♯minor and Cminor program states. We can access the size occupied by the variables of the function corresponding to a grame `f` with the `sz_vars` function; and the size of the stack block is accessed by the `stack_size` function. Each rectangle represents a new stack frame, either as a set of local variables (in C♯minor) or as a stack block (in Cminor). The height of the stack frames represents their size, and the size of the memory is the cumulated size of all the stack frames. The information that the `size_mem_preserved` predicates captures is depicted by the dashed lines: it remembers the relative size of the memories for each frame in the call stack.

At function entry, we need to add a new frame to `size_mem_preserved`, which is achieved through simple reasoning about the size of the memory after allocations. At function exit, we can use the `size_mem_preserved` fact to prove the `mi_size_mem` property of injections.

11 Related Work

Several works have proposed different memory models for the analysis of C programs. Among them is the work of Norrish [30], who gives a semantics for C using a concrete memory model (i.e. the memory is a mere array from concrete addresses to bytes). Reasoning about memory operations in those terms is difficult. Tuch *et al.* [31] use separation logic to make this reasoning tractable. VCC [11] transforms C programs annotated with specifications and function contracts into verification conditions [10], that are subsequently solved used the Z3 SMT solver. This work aims at verifying a hypervisor. It uses a typed memory model where the memory is a mapping from typed pointers to structured C values. Again, pointers are mere integers. This memory model is not formally verified.

Using Isabelle/HOL, Autocorres [13,14] constructs provably correct abstractions of C programs. The abstractions are expressed in a monadic style. Since the abstraction is correct, any property derived from the abstraction also holds for the C program. The memory models of VCC [11] and Autocorres [14] ensure separation properties of pointers for high-level code and are complete with respect to the concrete memory model. The CompCert model [26] has disjoint blocks by construction, therefore no logic is required to ensure separation properties. For our symbolic extension, the completeness (and correctness) of the normalisation is defined with respect to a concrete memory model and therefore allows reasoning over low-level idioms.

Several formal semantics of C are defined over a block-based memory model. Ellison and Roşu [12] define `kcc`, a C semantics based on the $\mathbb{K}$ framework that encodes semantic rules as rewriting systems. Their semantics follows closely the C standard and is executable. Most recently [15], they extend their work to give what they call a *negative semantics* of C, that allows to detect all kinds of undefined behaviours listed in the standard when they occur. Their aim is *to detect maximally portable, strictly-conforming programs*, and is therefore orthogonal to ours, which is centered around formal proofs of compiler transformations.

The CompCert C semantics [7] provides the specification for the correctness of the CompCert compiler [24]. CompCert is used to compile safety critical embedded systems [2] and the semantics departs from the ISO C standard to capture existing practices. For example, signed integer arithmetic is defined to wrap around modulo in case of overflow. Another example is related to sequence points. In C, it is undefined behaviour to access the same object several times between two sequence points, but CompCert assigns an arbitrary evaluation order and defines the semantics of such programs. Our work goes further in that direction and defines semantics of even more non-conforming programs.

Krebbers *et al.* extend the CompCert semantics but aim at being as close as possible to the C standard [22]; they formalise sequence points in non-deterministic programs [21] and strict aliasing restrictions in `union` types of C11 [20]. This is orthogonal to the focus of our semantics, which gives a meaning to implementation defined low-level pointer arithmetic and models bit-fields.

Most recently, Kang *et al.* [19] propose a formal memory model for a C-like language which allows optimisations in the presence of casts between integers and pointers. Pointers are kept logical until they are cast to integers, then a concrete address is non-deterministically assigned to the block of the pointer. This means that a pointer-integer cast may fail if no chunk of memory is available. They have proof principles that enable them to prove for two given programs that they are equivalent, or that one is a correct optimisation of the other, however they do not have a proven compiler/optimizer, which is what we aim at. Their model also lacks an essential property of CompCert's memory model: determinism. For instance, with a fully concrete memory model, allocating a memory chunk returns a non-deterministic pointer – one of the many that does not overlap with an already allocated chunk. In CompCert, the allocation returns a block (merely an identifier, not an actual address) that is computed in a deterministic way. As discussed in Section 3.1, determinism is instrumental for the simulation proofs of the compiler passes and its absence is a show stopper. Indeed, the final theorem of CompCert is a backward simulation. However, forward simulations are easier

to reason about and can be transformed into backward simulations provided that the input language is deterministic.

Carbonneaux *et al.* [8] propose *Quantitative CompCert*. This is an extension of CompCert that gives additional guarantees about the resource consumption of programs compiled by CompCert. For example, they give formal bounds on the stack usage of C programs. They compute these bounds thanks to a dedicated Hoare logic at the Clight level, and prove that these bounds are preserved by the compilation. Our model follows this direction because the compilation makes the memory usage of programs decrease, as we discussed in Section 9.2.

12 Conclusion

This work is a milestone towards a CompCert compiler proved correct with respect to a more concrete memory model. Our formal development adds about 20000 lines of Coq to the existing CompCert memory model and another 20000 lines for the proofs of the front-end. A side-product of our work is that we have uncovered and fixed a problem in the existing semantics of the comparison with the `NULL` pointer. Because we tested the CompCert C semantics with a low-level memory model, we strongly believe that this is the very last remaining bug that can be found at this semantics level. We also prove that the front-end of CompCert can be adapted to our refined memory model. The proof effort is substantial: the proof script for our new memory model is twice as big as the existing proof script. The modifications of the front-end are less invasive because the proof of compiler passes heavily rely on the interface of the memory model.

As future work, we shall study how to adapt the back-end of CompCert. We are confident that program optimisations based on static analyses will not be problematic. We expect the transformations to still be sound with the caveat that static analyses might require minor adjustments to accommodate for our more defined semantics. A remaining challenge is register allocation which may allocate additional memory during the spilling phase. An approach to solve this issue is to use the extra-memory that is available due to our pessimistic construction of stack frames. In spite of the remaining difficulties, we believe that the full CompCert compiler can be ported to our novel memory model. This would improve further the confidence in the generated code.

References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, chap. CVC4, pp. 171–177. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). DOI 10.1007/978-3-642-22110-1_14. URL `http://dx.doi.org/10.1007/978-3-642-22110-1_14`
2. Bedin França, R., Blazy, S., Favre-Felix, D., Leroy, X., Pantel, M., Souyris, J.: Formally verified optimizing compilation in ACG-based flight control software. In: ERTS2 (2012). URL `https://hal.inria.fr/hal-00653367`
3. Bernstein, D.J., Lange, T., Schwabe, P.: The Security Impact of a New Cryptographic Library. In: LATINCRYPT'12, *LNCS*, vol. 7533, pp. 159–176. Springer (2012)
4. Besson, F., Blazy, S., Wilke, P.: Companion website with Coq development. URL: http://www.irisa.fr/celtique/ext/frontend-symbolic

5. Besson, F., Blazy, S., Wilke, P.: A precise and abstract memory model for C using symbolic values. In: APLAS, *LNCS*, vol. 8858 (2014)
6. Blazy, S.: Experiments in validating formal semantics for C. In: C/C++ Verification Workshop. Raboud University Nijmegen report ICIS-R07015 (2007)
7. Blazy, S., Leroy, X.: Mechanized Semantics for the Clight Subset of the C Language. J. Automated Reasoning **43**(3) (2009)
8. Carbonneaux, Q., Hoffmann, J., Ramananandro, T., Shao, Z.: End-to-end verification of stack-space bounds for C programs. In: PLDI '14, p. 30. ACM (2014)
9. Clements, A.T., Kaashoek, M.F., Zeldovich, N., Morris, R.T., Kohler, E.: The scalable commutativity rule: designing scalable software for multicore processors. In: SOSP. ACM (2013)
10. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., al.: VCC: A Practical System for Verifying Concurrent C. In: TPHOLs, *LNCS*, vol. 5674. Springer (2009)
11. Cohen, E., Moskal, M., Tobies, S., Schulte, W.: A Precise Yet Efficient Memory Model For C. ENTCS **254** (2009)
12. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: POPL. ACM (2012)
13. Greenaway, D., Andronick, J., Klein, G.: Bridging the Gap: Automatic Verified Abstraction of C. In: ITP, *LNCS*, vol. 7406. Springer (2012)
14. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: formal verification of C code without the pain. In: PLDI. ACM (2014)
15. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of c. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15), pp. 336–345. ACM (2015). DOI http://dx.doi.org/10.1145/2813885.2737979
16. ISO: C Standard 1999. Tech. rep., ISO (1999). URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`
17. ISO: C Standard 2011. Tech. rep., ISO (1999). URL `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf`
18. Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: POPL (2015). DOI 10.1145/2676726.2676966. URL `http://doi.acm.org/10.1145/2676726.2676966`
19. Kang, J., Hur, C.K., Mansky, W., Garbuzov, D., Zdancewic, S., Vafeiadis, V.: A formal C memory model supporting integer-pointer casts. In: PLDI. ACM (2015)
20. Krebbers, R.: Aliasing restrictions of C11 formalized in Coq. In: CPP, *LNCS*, vol. 8307. Springer (2013). DOI 10.1007/978-3-319-03545-1_4. URL `http://dx.doi.org/10.1007/978-3-319-03545-1_4`
21. Krebbers, R.: An operational and axiomatic semantics for non-determinism and sequence points in C. In: POPL. ACM (2014)
22. Krebbers, R., Leroy, X., Wiedijk, F.: Formal C semantics: Compcert and the C standard. In: ITP 2014, *LNCS*, vol. 8558. Springer (2014)
23. Lee, D.: A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`
24. Leroy, X.: Formal verification of a realistic compiler. C. ACM **52**(7) (2009). URL `http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf`
25. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009). DOI 10.1007/s10817-009-9155-4. URL `http://dx.doi.org/10.1007/s10817-009-9155-4`
26. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert memory model. In: Program Logics for Certified Compilers. Cambridge University Press (2014). URL `http://hal.inria.fr/hal-00905435`
27. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. J. Automated Reasoning **41**(1) (2008). URL `http://gallium.inria.fr/~xleroy/publi/memory-model-journal.pdf`
28. MIRA Ltd: MISRA-C:2004 Guidelines for the use of the C language in critical systems (2004). URL `www.misra.org.uk`
29. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS, *LNCS*, vol. 4963. Springer (2008)
30. Norrish, M.: C formalised in HOL. Ph.D. thesis, University of Cambridge (1998)
31. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: POPL. ACM (2007)

32. Wang, X., Chen, H., Cheung, A., Jia, Z., Zeldovich, N., Kaashoek, M.: Undefined behavior: What happened to my code? In: APSYS '12 (2012)
33. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: PLDI. ACM (2011)